U.PORTO

MESTRADO
MULTIMÉDIA - ESPECIALIZAÇÃO EM MÚSICA INTERATIVA E DESIGN DE SOM

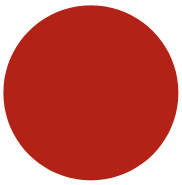
Recreating tracker music sequencers in modern videogames: an integrated model approach for adaptive music

Valter Miguel Alves Abreu

M
2018

FACULDADES PARTICIPANTES:

FACULDADE DE ENGENHARIA
FACULDADE DE BELAS ARTES
FACULDADE DE CIÊNCIAS
FACULDADE DE ECONOMIA
FACULDADE DE LETRAS



FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Recreating tracker music sequencers in modern video-games: an integrated model approach for adaptive music

Valter Miguel Alves Abreu

DISSERTATION

Mestrado em Multimédia

Supervisor: João Tiago Neto Jacob

Second Supervisor: Eduardo Magalhães

July 17, 2018

Recreating tracker music sequencers in modern video-games: an integrated model approach for adaptive music

Valter Miguel Alves Abreu

Mestrado em Multimédia

Aprovado em provas públicas pelo Júri:

Presidente: Rui Penha (Professor Auxiliar)

Vogal Externo: Nelson Troca Zagalo (Professor Associado)

Orientador: João Tiago Neto Jacob (Investigador Auxiliar)

July 17, 2018

Abstract

Music plays a very important role in video-games contributing to the game's atmosphere and immersion. It also operates at a cognitive level, namely in memory triggering feelings such as nostalgia. Theme songs from several classics, both from gameplay or the main menu are still remembered today and will probably never be forgotten.

During video-games' evolution, several technological iterations and increments arose, both for graphics and audio, as means to achieve better quality for the final product. Thus, many editing tools for adaptive audio or soundtracks are available today and pursuing an inherent concern: the game's music should ideally be dynamic, in the sense that it must adapt to the player's actions, as opposed to something that is static and perceived as continuously looping.

A software that enables this kind of approach with ease is a kind of music sequencers, commonly named *Trackers*, that are somewhat underused/obsolete these days. This technology, which comes from the Commodore Amiga times, brought many advantages over previous solutions and offers a very small storage footprint. With this, musicians and programmers found a new method for music creation and expression, that marked many gamers during the 90's. Nowadays there's little development and support for these sequencers, due to new solutions appearing with a lot of potential, although these don't share many of the trackers' advantages, thus being unable to completely replace the tracker.

This study focuses on the development of an architecture that can play music composed on a tracker, using current resources and frameworks, offering compatibility with modern game engines and allowing the expansion of the traditional trackers workflow and possibilities.

A prototype was developed based on an existing and updated application, from which its features and file formats were studied. Following this phase, the prototype was tested on the engine with a game developed specifically for the purpose, where several results were collected. From the observation and analysis of the results, it was possible to create a song with tempo control, instruments, musical patterns and many other elements. This track can be stored in a group of files with a relatively small size compared to other existing solutions.

Keywords: Trackers, Video-games, Adaptive, Music

Resumo

A música tem um papel fundamental nos vídeo-jogos, contribuindo para o ambiente e imersão do jogo, mas também pelas suas caraterísticas nostálgicas. Diversos clássicos ficaram marcados pelos temas que reproduziam no ecrã inicial ou nos primeiros minutos jogáveis, que ainda hoje são relembrados e dificilmente serão esquecidos.

Acompanhando a evolução, várias abordagens tecnológicas foram surgindo, no domínio dos gráficos e do áudio, de forma a obter mais qualidade no produto final. Como resultado, existem hoje várias ferramentas para criação de áudio ou de composição da banda sonora de forma adaptativa. No âmbito musical existe uma questão intrínseca: a música de um jogo deve ser idealmente dinâmica, isto é, cada secção da composição deve acompanhar as ações do jogador, ao contrário de algo estático que se repete continuamente.

Um dos *software* que permitem esta abordagem com facilidade é um tipo de sequenciadores, frequentemente denominados por *Trackers*, que estão algo esquecidos nos dias de hoje. Esta tecnologia, proveniente dos tempos do Commodore Amiga, trouxe vantagens sobre as anteriores com ficheiros que ocupavam muito pouco espaço em disco. Assim, a comunidade de músicos e programadores encontrou uma nova forma de criação e expressão musical, que acabou por marcar a década de 90. Atualmente existe pouco desenvolvimento e suporte para estes sequenciadores, devido ao aparecimento de outras soluções com enorme potencial e versatilidade, apesar de estas não partilharem o mesmo tipo de vantagens dos *trackers*, sendo difícil substituí-los por completo.

Este estudo foca-se no desenvolvimento de uma arquitetura capaz de reproduzir música composta num *tracker*, recorrendo a ferramentas e soluções atuais, de forma a que seja expansível e compatível com motores de jogo modernos.

O projeto foi desenvolvido baseado numa aplicação existente e atualizada, da qual foram estudadas as suas caraterísticas e particularidades, bem como os formatos de ficheiro associados. Depois desta fase, o protótipo foi testado no motor com um jogo desenvolvido para o propósito, onde foram conferidos os resultados esperados. Analisando estes resultados, tornou-se possível criar uma música com controlo de tempo, instrumentos, padrões musicais e muitos outros elementos, armazenada num conjunto de ficheiros com um tamanho substancialmente mais reduzido que a maioria das soluções existentes.

Keywords: Trackers, Videojogos, Adaptativa, Música

Agradecimentos

Aos orientadores João Jacob e Eduardo Magalhães por todo o tempo despendido na motivação, planeamento, sugestões e propostas de soluções nos diversos momentos do desenvolvimento.

Ao José Raimundo pela disponibilidade em partilhar o seu projeto, com a qual a colaboração resultou numa prova de conceito fundamental para a validação das propostas apresentadas.

Aos meus amigos e colegas Nuno Loureiro e Alonso Torres Matarrita pelo apoio e ideias que em várias alturas resultaram em soluções indispensáveis em pontos essenciais do desenvolvimento.

Aos meus pais, ao meu irmão Rui e em especial à Margarida por toda a valorização e apoio prestado.

A todos os meus amigos pelo companheirismo e incentivo.

Valter Abreu

Contents

List of Figures

List of Tables

Abbreviations

DAW	Digital Audio Workstation
MIDI	Musical Instrument Digital Interface
OSC	Open Sound Control
PD	Pure Data
FPS	First Person Shooter
VST	Virtual Studio Technology
VSTi	Virtual Studio Technology Instrument

Introduction

With the emerging of the first person shooters during the 90's, trackers were beginning to be used as a tool to create adaptive music with great results, as composers could create high-quality music stored in a considerable small-sized file.

This study falls within the context of dynamic soundtracks for computer games, using a relatively old music composition system, often called *Tracker*. This type of software debuted in classic personal computers such as the Commodore Amiga and IBM PCs, being later recreated resulting in several versions in this decade. A big part of the tracker development occurred in the so-called *demoscene*[1], a community of artists and developers that shared their work as demonstrations. Usually these people ended up starting their own companies or working for popular games at the time.

This movement might had an influence over modern middleware, since it is used to structure several audio files previously recorded on a DAW[2]. This enables the sound artist to program different behaviours for the audio and then save it as a package to be controlled by in-game events. However, trackers are slightly different since they do not use pre-recorded (static) music, instead they just carry information about tempo, pitch, volume and patterns. It is similar to a MIDI[3] file but also stores the samples it uses to play the information.

[1] http://www.demoscene.info/

[2] DAW is an abbreviation for Digital Audio Workstation. This is the most common software that composers and audio producers use to compose, record, mix and master songs.

[3] MIDI stands for Musical Instrument Digital Interface and it is the standard protocol for communication between devices. In classic videogames this protocol was used in order to save disk space since a MIDI file only stores data related to notes, velocity and other important music information. The downside is that the sound depends on the machine playing the file, that would obviously show a result different from what was intended by the composer.

Motivation

Trackers offer a particular way of composing music, having its own style and expression characteristics: "the rapid editing interaction (low viscosity) and fast feedback cycle (progressive evaluation) supports a high degree of liveness, enabling sketching and flow". [Nash, 2014]. However, with the technological evolution of music composing systems for video-games, the software became less common.

This software also provides flexibility and scalability when compared with the current typical solutions. For example, when game engines had full support for module files, which was the case with the first version of Unreal Engine, composers only needed to create their songs and send them to the programmers, no other tool or resource was needed during the process in order to get an interactive piece of music actually working: "the MOD file can be constructed of detached numbered musical patterns that can be juggled and reordered as needed. This enables the audio programmer to assign interactive behaviors to these patterns with relative ease." [Phillips, 2014].

When analysing other approaches, these can be in many ways more complex since the composer needs to create his music in a DAW, then export it to a middleware solution and then give it to the programmers to use it in the game engine. These are obviously very powerful solutions but it can be very difficult to establish a middle ground between musicians and programmers. In addition, it is not always necessary to use a very complex structure, due to budget or time related concerns which imply other faster and effective methods.

Having a new tool supporting classic methods of doing music for games can be extremely positive and valuable, people who still choose the tracker to compose their songs could finally have a way of implementing them in a game. "There is a substantial amount of artists creating music using trackers. (...) the majority have been making soundtracks on the demo scene for the past 20+ years." [Schweitz, 2010].

Problems, hypotheses and research goals

Looking at the most popular modern game engines, it can be confirmed that Unity Engine supports legacy tracker file formats and Unreal Engine 4 doesn't. "Unity supports the four most common module file formats, namely Impulse Tracker (.it), Scream Tracker (.s3m), Extended Module File Format (.xm), and the original Module File Format (.mod)." [Unity Documentation, 2017]. However, after conducting some basic preliminary tests, which will be discussed later, conclusions showed that Unity only allows the playback feature and no further tracker manipulation. For example: an instrument cannot be muted nor a specific pattern can be chosen to play.

Despite the lack of full support by the major game engines, trackers can be very powerful since current solutions for adaptive music are not as versatile and require many workarounds to get close to a result that would be simply achieved with a module file. On the other hand, file size footprint and CPU usage can be advantages in most handheld devices, which is something trackers may

guarantee. Module music is a performance, it has its own expression and it is highly manipulable. Rendered audio however, even with solutions to make it adaptive, does not have this potential.

Trackers nowadays are being developed mostly by the open-source community and there seems to be very little to no commercial interest regarding this subject. In this sense and according to the aforementioned problematics, trackers, when used as a tool to create dynamic music for games can solve the issues of non-intuitive approaches, heavy-weighted static audio files and expressiveness. The aforesaid issues point to the following research questions:

- **RQ1**. Can trackers be used for adaptive music generation in modern game development?
- **RQ2**. Are trackers a valuable strategy for adaptive music generation?
- **RQ3**. What benefits does using a tracker provide in terms of resource usage?

Main goals:

- The first main objective of this research is to propose an audio platform capable of playing module files in an interactive way.
- Enable the integration of this model in a modern game engine. For testing the tracker music player, original songs will be composed in order to explore different musical strategies.
- Evaluate the benefits of using module files in terms of resource usage and understand its limitations and advantages. This prototype should run in mobile platforms.

Research methodology

The methodology was designed with a plan consisting in a state of the art research in order to review different concepts, a work model definition to organize the tasks for the development stages, a prototype implementation and the validation of its results. This work plan is shown in figure 1.

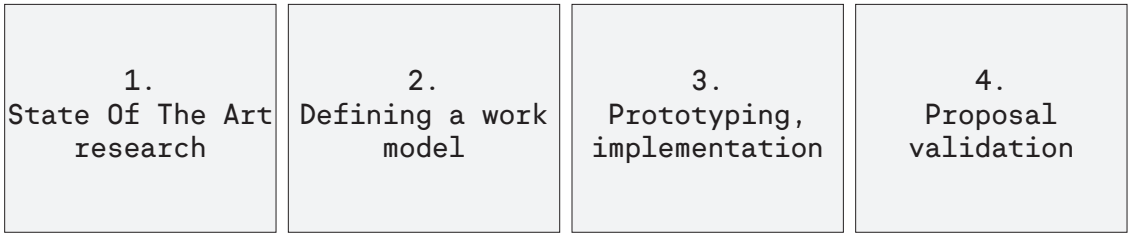


Figure 1: The research methodology.

The proposed prototype was made using a data flow[4] environment, in this case Pure Data. This software has proven to be very suitable because it is more familiar for musicians and capable enough to build a system as complex as this. Composer Leonard Paul has used it before, "he used oscillators, variable delay lines and granular synthesis patches in Pure Data to generate a dynamic soundtrack that could react to different game states".[Procedural Audio Now!, 2015].

Another related project that serves as a good example is the audio engine barelyMusician:

> barelyMusician is a comprehensive music composition tool, capable of generating and manipulating audio samples and musical parameters in real-time in order to create smooth transitions between musical patterns that portray varying emotional states and moods that may be evident during gameplay. [Gungormusler et al., 2015]

The fact that PD's library can be implemented in various forms to create a plug-in for different platforms makes it easier to achieve the main goal of this research. Along with this, different adaptive music strategies should be collected. These need to be strategies applied in games that meet the objectives of this study in order to make a preliminary analysis. After having the prototype ready, it will then be implemented in Unity Engine. This particular game engine was chosen for having its wide user base and to fill the gap in the tracker files support. The last task would be to test the game demo with real users in order to draw conclusions in terms of performance hit and audio quality.

[4] More info about Data Flow is presented in the Adaptive Music section.

Chapter 1

State Of The Art

1.1 Tracker Music

There are several ways to compose music these days: from real acoustic recordings to digital and electronic instruments, or from big orchestras to minimal soundtracks. The possibilities are immense, spanning into different genres, compositional approaches and even artistic movements that are associated with video-games' music.

The cultural side of this universe is present in several niches, but also in very large-scaled communities. There are genres such as the *Chiptune*, which is characterized by the sound of old computer and console games, often done with hardware from that time by underground musicians. On the other hand there are also highly produced soundtracks, mostly for AAA games[1] which are performed live in events for large publics, such as The Game Awards[2] in Los Angels, USA, or the Game Music Festival[3] in Wrocław, Poland.

In other times, home computers and consoles had much more limitations when comparing to today's standards, therefore the tendencies were different. During the late 80's an artistic sub-culture was growing: the *Demoscene*. This is known as an international group with the goal of producing digital performances that could demonstrate several skills including music, visuals or programming. The Tracker Music movement emerged from this sub-culture, and it focused on — at the time — a new software for home personal computers, often called *Tracker*.

Essentially, trackers are "step-time composition programs, popularized by the now-15-year old Amiga PC, that competed and won over the IBM PC-based General MIDI music technology for games." [Brandon, 2005, 27]. The software introduced new possibilities and advantages, due to both its technological aspects and characteristic sound.

[1] "AAA Game" is a typical definition for a video-game with a very high budget and production standards.
[2] http://thegameawards.com/
[3] http://gmfest.com/

Trackers behave in a similar form as MIDI in a sense that the music file is only information, however, it can also store audio samples along with the music score. "Both formats treat music as a data file, the contents of which include the musical performance and an accompanying sound bank containing the instruments to be triggered by that performance." [Phillips, 2014].

Early history

When going back to classic gaming platforms, there wasn't really a standard for music composition. For personal computers there was only General MIDI-based solutions which were very good, but they relied on the soundcard the composer was using. For example, if a person sent his song to another, the other might not be able to listen to it the same way as it was intended by the composer, because the hardware was not the same. Videogame consoles didn't have these problems because the hardware never changed: If someone bought a NES or a Genesis console, they would always have the same hardware.

In fact, games that were released for several platforms usually had different music. Doom (1993) is an example of this, as the Nintendo 64 version (released later in 1997) featured a completely new soundtrack. The same happened with Descent (1995) as both PC and PlayStation versions did not feature the same music, changing from MIDI to CD Audio, possibly because Sony was popularizing the CD format.

While the aforementioned solutions were being used by composers, a more reliable and effective music sequencing program was released, The Ultimate Soundtracker, shown in figure 1.1. This software was developed for the Commodore Amiga by Karsten Obarski, a composer and programmer at the german video-game company reLINE [Collins, 2008, 58], and also a member of the demoscene. Tracker music was born at this point, having its natural evolution afterwards, which resulted in more enriched computer programs.

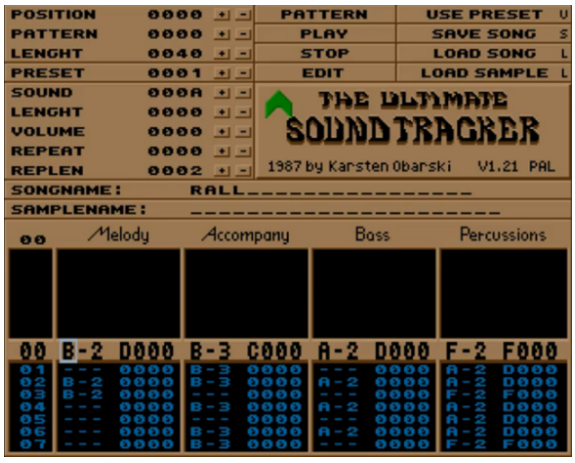


Figure 1.1: The Ultimate Soundtracker (1987)

Well-known game companies started to hire people from the demoscene, including musicians. Epic MegaGames[4] was one of them, which had a big appropriation of tracker music for their games, one of the most relevant being Unreal (1997), composed by Straylight Productions. "Unreal was able to have very truly unique and high-quality sound. (...) the composers at Straylight Productions were able to loop sections of music devoted to as many gameplay situations as were needed, from suspense to action to death to exploration." [Brandon, 2005, 89-90].

[4]Now known as Epic Games, the Unreal Engine developer.

Straylight Productions was founded by Alexander Brandon, Andrew Sega and Dan Gardopée, which are some of the best-known composers in the tracker music community. Besides the Unreal series of games, they worked for other Epic titles such as Jazz Jackrabbit 2 (1998) and later on Ion Storm's Deus Ex (2000). It was then after this period that tracker music was starting to be less common on games, as easier solutions were emerging, namely rendered audio formats.

With the propagation of CDs and other digital audio standards such as the mp3, video-games were somehow abandoning classic sequenced track formats, such as MIDI and module files. In fact, mainstream music artists were starting having their music featured on games, which was the case of id Software's Quake (1995), with a soundtrack composed by the industrial act Nine Inch Nails. PlayStation had many titles where audio was recorded on the CD and streamed in-game, WipeOut 2097 (1996), Kula World (1998) or Metal Gear Solid (1998) to name a few. Most of the times people could actually insert the game's disc into a stereo system and playback the music because it was recognized as a CD Audio. A case that shows the migration from sequenced formats to recorded audio is Deus Ex (2000) which features a soundtrack composed with trackers but in the PlayStation 2 port the main theme was re-recorded by a live orchestra.

Currently trackers are being developed and maintained mostly by the open-source community, with some active composers using it. Music associated with the software can sometimes be found in mobile platforms. "Today, we'll find both MIDI and MODs sporadically used in games created for some handheld devices that have memory limitations, making it difficult for them to incorporate an audio-only musical score." [Phillips, 2014].

Figure 1.2: Plants vs. Zombies (2009)

Plants vs. Zombies (2009), shown in figure 1.2, is a good example of a mobile game featuring tracker music. The soundtrack in this game is a looping song, which is constantly introducing new instruments and variations to the mix, following the increasing difficulty. It is a frequent technique that gives the player the idea of progress, which helps keeping him attached to the game.

Software evolution

The first generation started with the already mentioned Ultimate Soundtracker. The filetype associated to this software is the .MOD, often called *module file*, and it's still one of the most popular Tracker formats. Development of new software based on this format continued, naturally moving to the MS-DOS operating system, during the beginning of the 90's.

A second generation of multi-platform and improved trackers started to appear, mainly for computer operating systems such as Windows and classic Mac Os. These were very popular in PC gaming up until the early 2000's, with many great titles using it such as Deus Ex (2000). After

this, as disk storage was starting to be no longer a problem, the use of trackers almost disappeared since composers switched to other digital audio solutions.

Currently trackers are being developed and maintained mostly by the open-source community. One of the most popular is OpenMPT, which offers support for all legacy formats and has its own native format. According to the manual, "It comes with a built-in sampler but may also make use of external sound generators, including external MIDI synthesizers and “virtual synthesizers” called VST instruments".[5] MilkyTracker is another one that is also largely used, but it is essentially a recreation of the DOS software FastTracker II and is specified as a second-generation Tracker by the developers.[6]

Among these and many others available, there is a commercial one called Renoise. This particular software is an hybrid according to the developers: "Renoise is a Digital Audio Workstation (DAW) with a refreshing twist. It lets you record, compose, edit, process and render production-quality audio using a tracker-based approach".[7] This is the natural evolution for trackers as Renoise offers many improvements over the classic ones, granting its place on the music production market. It is able to import all legacy formats and has its own, open native file format.[8]

1.2 Adaptive Music

The term *adaptive* refers to the way of designing sound or composing a piece of music with a nonlinear structure. In short, “it is understood as any audio that is nonlinear or nonreactive in a game. For example, in a linear medium such as a cut scene, a movie, or a full-motion video clip, the audio is linear: It doesn’t change no matter how many times the medium is played.” [Brandon, 2005, 85]. Essentially, the game’s storytelling can be reinforced by variations of the music that will fit different game states, as seen in figure 1.3.

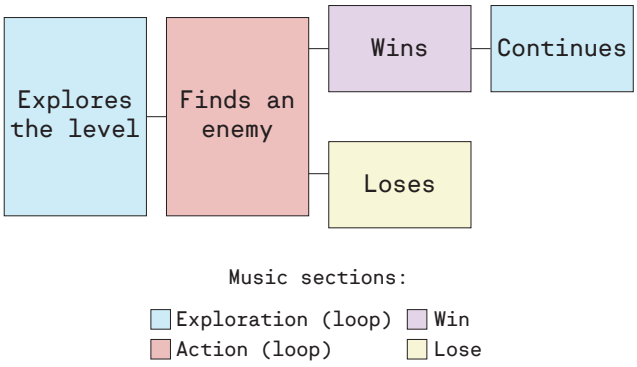


Figure 1.3: Simple game/music states diagram

Adaptive music is present in most video-games, being more or less dynamic in each case. It is being made since many classic games, one example is Super Mario Bros (1985), figure 1.4, which has done some interesting techniques at the time. "(...) in Super Mario Bros tempo is increased when the player is playing well and he or she has collected enough power-ups." [Velardo, 2017].

[5] OpenMPT Manual: https://wiki.openmpt.org/Manual:_About_OpenMPT
[6] About MilkyTracker: http://milkytracker.titandemo.org/about/
[7] About Renoise: https://www.renoise.com/products/renoise
[8] This subject will be further discussed on the Solution Development chapter.

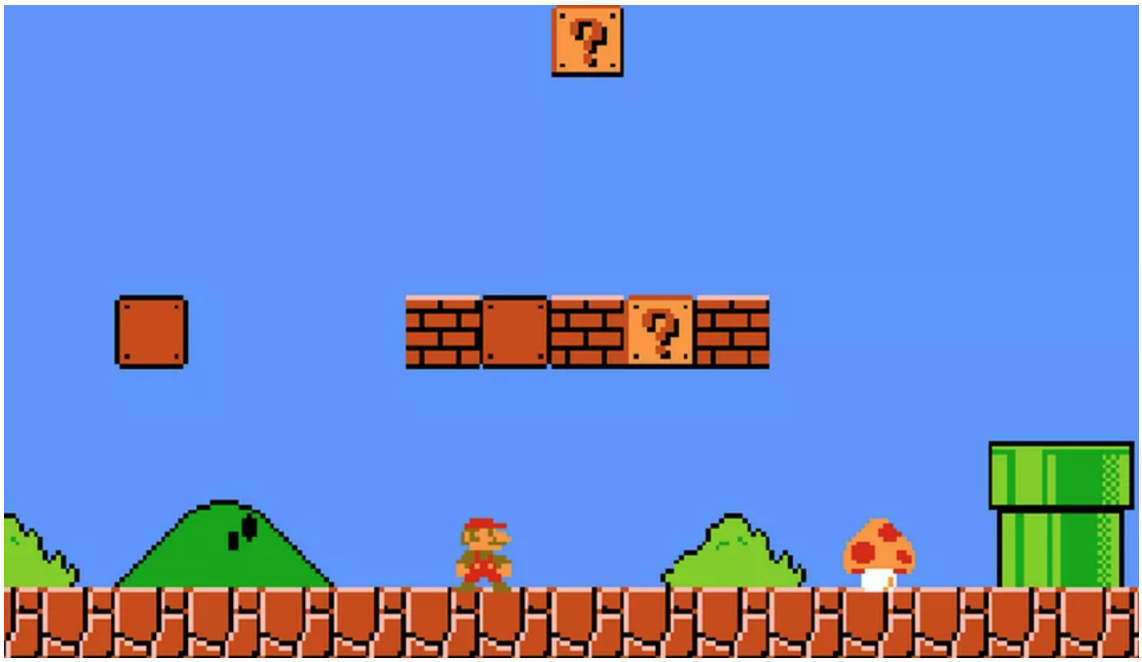


Figure 1.4: Super Mario Bros (1985).

In order to adapt, several musical parameters from the song can be mapped. These can be from loop sections alternation to re-orchestration (see 1.2). Several studies/approaches identify and explore compositional problematics related to harmonic and tonal content generation, such as Lexikon-Sonate (1992) by Karlheinz Essl. In the context of adaptive music these approaches may contribute to cope with the problem of the sense of repetition, variation, coherence as well as keeping the listener interested/attached to the game environment.

Some have experimented with the manipulation of different sound sources by composing pieces influenced by acousmatic music[9], which is the case of Limbo (2010), shown in figure 1.5. "Studying acousmatic music and soundscape composition in general have also served as an important inspiration in my audiovisual and interactive work, although the concepts associated with these genres are not directly applicable to audiovisual media." [Andersen, 2011]. The game's music is composed of ambient soundscapes, which makes it easier to introduce new audio layers with a seamless result, since it usually does not need to be attached to a BPM or other kinds of parameters. This way, the music is constantly varying without the player noticing it.

Figure 1.5: Limbo (2010).

[9] Acousmatic is a genre of electroacoustic music, where the performance is made through speakers, preventing the listener from viewing its source.

There are many ways to integrate dynamic music or sound design in video-games. A common way to do so is by using middleware. This is basically a program that is used after the composition/recording stage of the track/audio object, and before its implementation on the game engine. Examples of this are popular applications such as Audiokinetic's Wwise or FMOD by Firelight Technologies.

As an example, the composer creates his track on his preferred DAW and then he renders its sections and variations into different files. These files are then imported into the middleware where he can define different behaviours for the song, resulting in a well-structured audio object that can be used by the programmer on the game engine in order to play different sections of the song using in-game events. In addition, middleware "allows audio developers to prototype the integration of audio in a game simulation before the game is even finished." [Collins, 2008, 100].

Making rendered audio dynamic is not something entirely new, the PlayStation classic PaRappa the Rapper (1996), had a very interesting approach to adaptive audio, it focused on music and particularly on rhythm. Through quick-time events, the player would have to reproduce rhythmic patterns given by the game in order to succeed. The music then was constantly varying. Titles with similar characteristics to this are Audiosurf (2008), shown in figure 1.6, a game where "The shape, the speed, and the mood of each ride is determined by the song you choose"[10], and the popular Guitar Hero (2005), a game where audio files are streamed only when the player hits the notes displayed on screen.

Figure 1.6: Audiosurf (2008)

The horizontal and vertical approaches

Adaptive music can be accomplished in two different approaches, the horizontal and the vertical. The horizontal means that sections of the music are triggered according to the current game state, "Horizontal re-sequencing, or cell based music, can be considered as a method that enables the reorganisation of the sequence of pre-composed musical fragments, instead of this sequence being fixed in advance". [Van Nispen Tot Pannerden et al., 2011]. The vertical however, implies more interesting and complex manipulations of the music, it is more similar to mixing audio in real time, "vertical re-orchestration, utilising 'stems' (or tracks) that are mixed according to a specific game-play variable, tend to consume more disk space". [Van Nispen Tot Pannerden et al., 2011].

An example of a popular game which uses the vertical re-orchestration approach is Portal 2, shown in figure 1.7.

[10]Read more at http://store.steampowered.com/app/12900/AudioSurf/

"There are several cases where the music adds channels and complexity as you successfully solve portions of the puzzle, with each additional piece of music actually coming from the device that is participating in the activated game play mechanic." [Morasky, 2011].

Figure 1.7: Portal 2 (2011).

While vertical re-orchestration is based on layering different tracks, the horizontal re-sequencing approach can be done using different methods, shown in table 1.1 [Volk, 2017]:

Horizontal re-sequencing	
Methods	**Procedure**
Cross-fading	One music cue fades out while another music cue fades up. Commonly used in most games.
Phrase-branching	Waits for the current musical phrase to end before playing the next musical cue.
Musical demarcation branching	Allows the music cue to switch at a musical demarcation point such as a beat, or measure.
Bridge transition	Short musical cues are used to connect one musical cue with another.
Stinger-based sequencing	A series of stingers which are played back based on game events. These stingers do not connect with one another, but may overlap.

Table 1.1: Table showing the different horizontal re-sequencing methods.

Generative approach

Generative procedures are sometimes used to create musical patterns that are constantly adapting to the game. These can be very complex since the concept has its roots in philosophy, which defends that certain events do not have a cause, thus simply happening from nothingness. “The core philosophy of generative music is based on the idea of indeterminacy—the introduction of chance into the unfolding of a composition and the randomization of musical content for the purposes of rendering something that is constantly unique.” [Phillips, 2014].

It could be established here a relation with the concept of randomness, as anything that is random is not pre-determined or predictable. Generative music usually focuses on a semi-controlled chaos on a given algorithm, meaning that the raw material is not known (e.g. a set of randomly generated notes), but it will follow certain procedures and rules in order to create several different musical patterns that make sense with each other.

Randomly generated music comes from very old times, as many known composers built their own systems, in order to have a certain piece as a result. *Musikalisches Würfelspiel* (1792) is an example of one of these systems, and often known as being used by Mozart. It consists in a dice, with its numbers corresponding to a previously composed musical pattern, that when it is tossed the composer must use the respective pattern in his music.

Music of Changes (1951) by composer John Cage is often categorized as one of the best known indeterministic music works. The composition system was based on *I Ching*, an ancient Chinese book that influenced many individuals on decision-making processes.

Composer Brian Eno has done work with generative music as well, also contributing to videogames with his soundtrack for Spore (2008), figure 1.8, "Spore’s soundtrack is procedural, which means that it changes based on a gamer’s style of play (e.g. more aggressive play triggers more aggressive music) and that no two gaming sessions will sound exactly the same". [Maher, 2008].

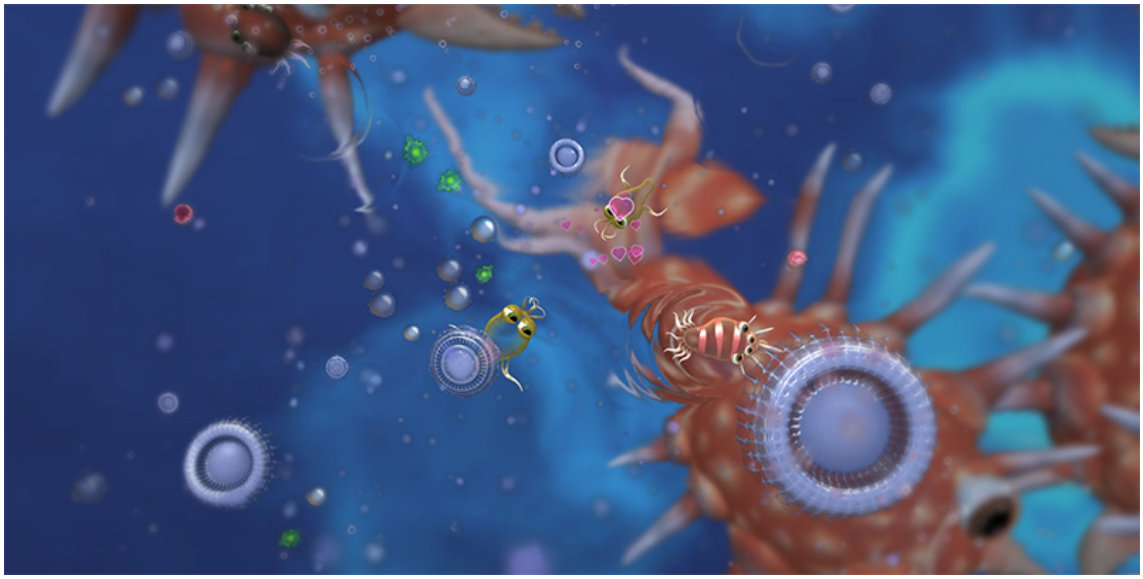


Figure 1.8: Spore (2008).

Data flow software plays an important role on these subjects as it enables the composer to program a complete music generation system the way he wants. This system will then communicate with the game engine through OSC[11] or MIDI protocols.

The most popular data flow applications are Max MSP and Pure Data. The latter is more useful in the case of video-games because it is open-source and it can be embedded in a game engine by compiling its library. A very interesting example for this is the game FRACT OSC (2014), in which the player explores a musical world where he can interact with several different oscillators and sound sources, as well as creating his own musical sequences. In the end the soundtrack will always have different results. Figure 1.9 shows the player interacting with an in-game music sequencer.

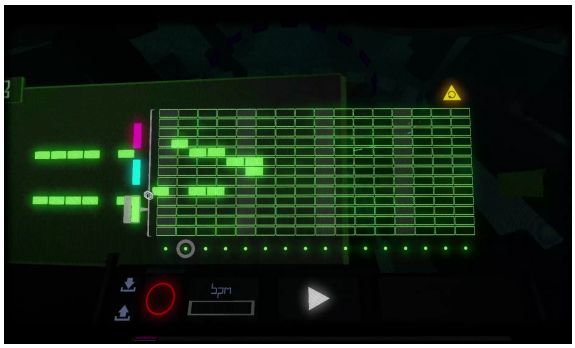


Figure 1.9: FRACT OSC (2014)

1.3 Summary

The first section of this chapter portrayed tracker music within the context of composition solutions, as well as the importance of the demoscene, where most of the work related to this movement was developed from both artistic and technological points of view. It was reviewed afterwards the evolution of the software.

The second section is related to the concept of adaptive music, which is something very present in video-games, as they are often focused in non-linear performances. An historical overview is presented here, where a connection with different music periods was made, in order to put it in context.

By reviewing the purpose behind these subjects, as well as their history and evolution, it was retained that there is an obvious relation between them. A tracker music sequencer enables the possibility to manipulate a composed song in various forms during gameplay, being a valuable solution to create an adaptive score. "The MOD format was easily adaptable to game sound, in that patterns (sequences) could be arranged to change volume, jump to other sequences, start or stop instruments, and so on." [Collins, 2008, 58].

[11] OSC is the abbreviation for Open Sound Control, a protocol for music communication which offers more versatility when comparing to MIDI. This protocol also offers communication over network.

Chapter 2

Methodology

In this chapter is presented the proposed methodology that was used on the development stage. During the outlining of this methodology, some preliminary tests had to be made, in order to decide which way to go during development. It was by the end of this stage that was decided what tools would be used, in order to embrace the more technical part of the project.

Using this method would enable testing some advantages of using module files and possibly establish some comparisons between this proposal and other approaches, such as using the same track rendered to audio files and then create an adaptive version of it using a middleware solution such as Elias[1] for example.

However, some difficulties were faced during this process: with Unreal Engine 4 it is not possible to use these formats and there's nothing associated with tracker music on the software's documentation. Continuing with the research, it was concluded that Unity offers support for the four main tracker music file formats, which opened doors for some initial tests.

Before beginning with the tests and the development of an architecture, it was important to study the main formats to understand their features, as well as to check their compatibility and integration with Unity.

2.1 Analysing trackers and their formats

Visually, a tracker's main window is similar to a spreadsheet, in a sense that it consists of columns and lines. Each column represents a track (sometimes called channel) and each line represents every possible beat, the number of lines is configurable for each pattern. The songwriting in a tracker is made using patterns (the equivalent to a group of measures in music notation), which can be used to easily repeat a certain part of the song or to create looping sections.

[1] Elias Software is a tool used to re-arrange several audio tracks to be played in different ways during gameplay. More information about this at: https://eliassoftware.com/

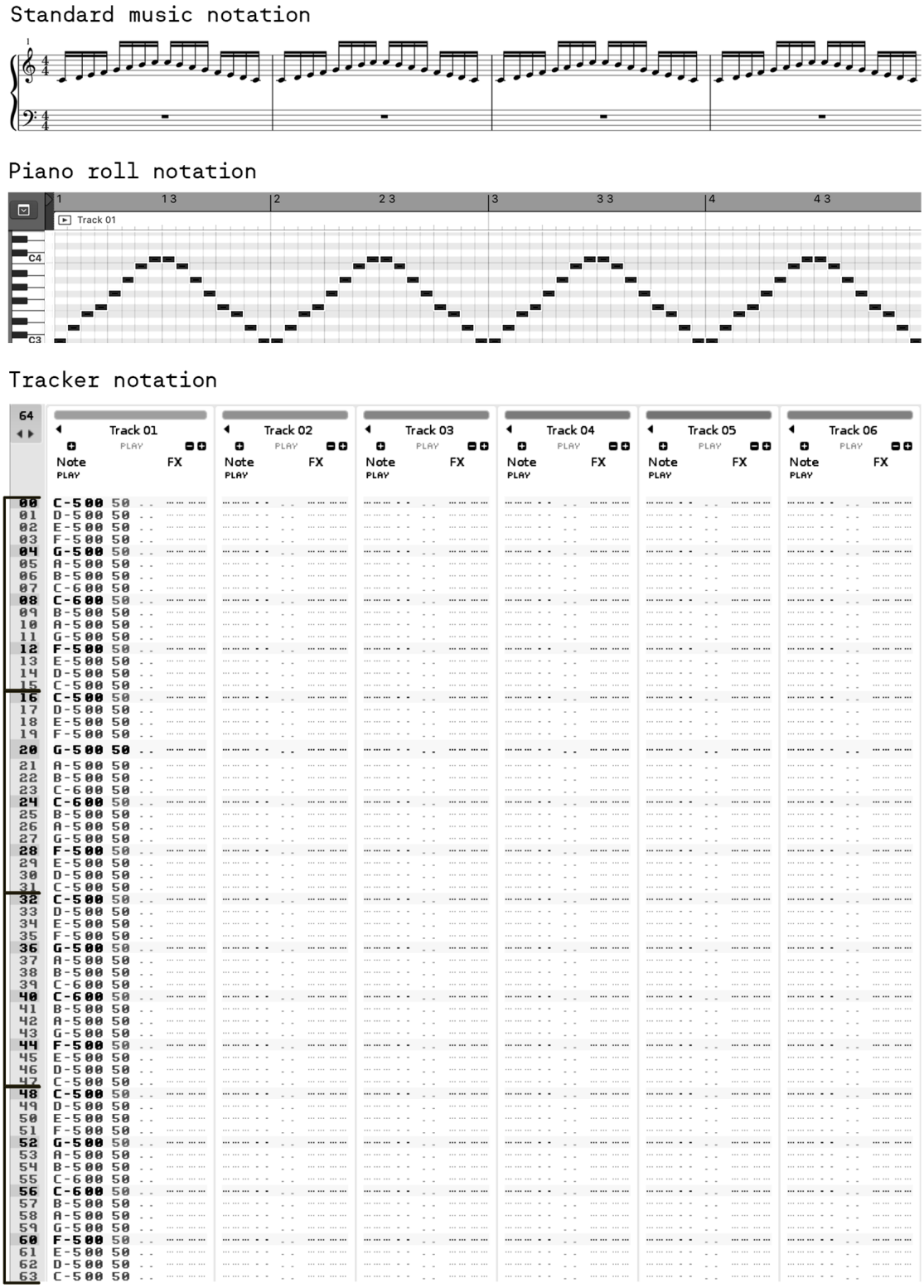


Figure 2.1: Beat display comparison between standard music notation, piano roll and tracker notation.

A pattern with 64 lines (which is typically the default) is the same, in standard music notation, as a set of four measures with the time signature of 4/4, where each line represents the duration of a sixteenth note (1/16). The metrics of a pattern can be adjusted by the user, in order to change the time signature or the number of lines, very similarly to a piano roll from any DAW. A comparison between each notation system is demonstrated on Figure 2.1.

In a traditional tracker, every line in a channel has four fields to write commands in. These are, by order, [Note - Instrument - Volume - Effect]. The note is written in standard notation (e.g. C-4), but the rest of the commands are usually hexadecimal values. In addition, the effects command has a letter before the value identifying the effect. The instrument is a value that represents an audio sample (or a set of samples) that are imported and used in the project.

Figure 2.2: Instruments table of a tracker music sequencer (Renoise in this case).

For example if a certain line reads "C-4 06", it means it will play the C-4 note on instrument number six. The instruments are most commonly shown in a table of the tracker's main window, as shown in Figure 2.2. These instruments are often known as patches; in modern trackers, such as OpenMPT or Renoise, a patch can be not only sample-based but also a VSTi[2] or another kind of sound generator.

Next is the volume command, this is a value from 00 to 7F, which represents a MIDI value from 0 to 127, and it is essentially the velocity of the note. Using the last example, when adding the volume command it would result in something like "C-4 06 50", where 50 is the volume of the note (80 in MIDI velocity). These values are obtained using the formula shown in equation 2.1.

$$DecimalNumber = Digit \times 16^{Position} \tag{2.1}$$

The number must be split into digits, with their positions counted right to left and starting at 0. Taking the number 50 from the previous example: 0 in position 0 is 0 and 5 in position 1 is 80. Hexadecimal is a base 16 numbering system, having six extra digits above the decimal system, which are represented by letters. With this system, numbers are counted from 0 to 9 and then from A to F, where A is 10 and F is 15.

The last field is used for the effects command. These effects are used for certain articulations, in order to create expression. These commands are usually values to control elements such as the envelope (e.g. apply a fade in effect on a note) or to produce articulations between notes, such as a *glissando*. This is where the difference between module file formats gets noticeable, as each extension has some specific effects that others might not have.

Module file formats can be divided in two major groups: the legacy formats – which include the classic and most common extensions, and the current formats – more related to modern and regularly updated software. Table 2.1 describes the different and most common formats[3].

[2] VSTi is the acronym used for Virtual Studio Technology Instrument.

[3] Most of the information was retrieved from: https://wiki.openmpt.org/Manual:_Module_formats

Legacy formats			
Extension	**Associated software**	**Platform**	**Features**
MOD	The Ultimate SoundTracker	Commodore Amiga	4 channels; Limited sample size; Few pattern commands
S3M	ScreamTracker 3	MS-DOS	Sample tuning; More channels; Volume command column; Compressed pattern data
XM	FastTracker 2	MS-DOS	Volume/panning envelopes for samples; Multi-sampled instruments; Panning (instrument level); Sample looping
IT	Impulse Tracker	MS-DOS	Instruments specify their samples' transposition; Apply different resonant filters to samples; New note actions; Panning (channel level)
Modern formats			
MPTM	OpenMPT	Windows	Custom sample tuning; Parameter control events; Multiple pattern sequences; Fractional tempos; Global resampling; VST support
XRNS	Renoise	Windows, macOs, Linux	Channel grouping; Multi-voiced channels (polyphony); Improved DSP + audio processing tools; Improved mixer, sampler and effects; Automation; VST/AU support; Rewire support

Table 2.1: Table with tracker formats specifications.

2.2 Preliminary Tests

After analysing the main formats and their features, it was time to start putting some of these subjects into practice, in order to define a methodology and a work plan. The main objective here was to use a song composed in a tracker and test it in some selected game engines, to check if they were compatible as well as to understand which features could be used.

A track in .IT format from an existing game (Deus Ex (2000) in this case) was chosen for testing purposes. After creating a new project in Unity and importing the file, it was possible to verify that the engine played the music as expected. Since this track contains different loop sections, as they are used in several moments from the level of the original game, the next step was to create a simple interface that would enable switching between patterns (from the main loop to the "action" loop for example).

However, there was no option to choose the current pattern, nor any function to call on the script that could do this. It was also tested, with no positive results, if it was possible to get multi-output from the file, as a way to control individual channels with Unity's own audio mixer. In order to check if this was a gap or just an hidden feature, some web searching had to be made. On Unity's online forums there were some users discussing this matter, as none of them could make programmatic changes to the music. It was later understood that the engine's audio library doesn't have any special feature to dynamically manipulate a tracker file, it only streams it as if it was a regular audio file. Afterwards, several searches were made as a way to find a solution for this problem. Some possibilities implied compiling external libraries that support the tracker format, but none of them was written in C#, which is the programming language Unity uses. Since there was no complete support for these formats, other possibilities had to be tested.

Analysing the current state of tracker music sequencers, there are two major pieces of software that are active in development, these are *OpenMPT* and *Renoise*. OpenMPT seemed like the best way to go, being an open-source software and a more natural evolution for the trackers as opposed to Renoise. However, OpenMPT's format (the .MPTM) is based on .IT and it is not supported in Unity. In the end, the only tool that was left which offered some possibilities was Renoise.

2.3 Analysing Renoise and the .XRNS format

Since Renoise supports legacy tracker formats, the next step was to open the previously used song. After some initial tests with the software, it was concluded that it is not possible to save the work in its original file format. Now, Renoise is meant to be used as a DAW, meaning the exports are generally in .WAV or similar. This could be used, but in the end it would be just a conversion from a tracker format to an audio format, which is not the purpose of this project. What is, in fact, useful is Renoise's own project file extension, the .XRNS. During some studies around this format, it was noticed that it works essentially as a .ZIP archive. Hence, using a file extraction utility such as The Unarchiver, several resources were retrieved: as seen in figure 2.3, the package contained a .XML, which describes everything related to the music, and a folder containing all the audio samples used for the different instruments inside the project. With a file type as easy to read as the XML, it should be possible to create an architecture that could extract all the musical data, interpret it and reproduce it using the original audio samples stored in the archive.

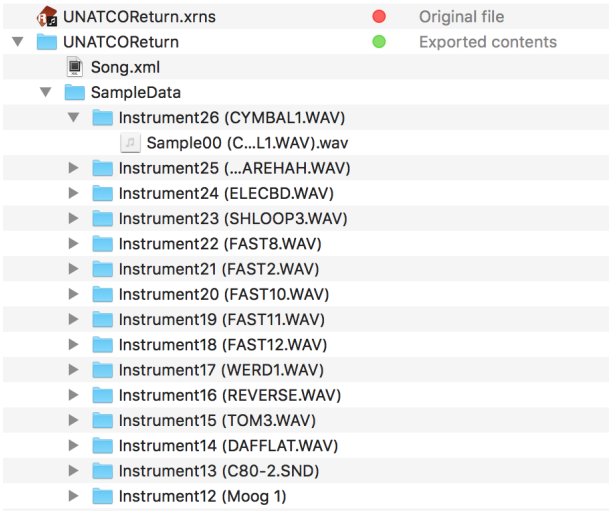


Figure 2.3: Files after exporting from the .XRNS

The .XML inside the Renoise file clearly describes every value from the song. The BPM, along with other global data, can be found right on the first lines of the file, as seen in figure 2.4.

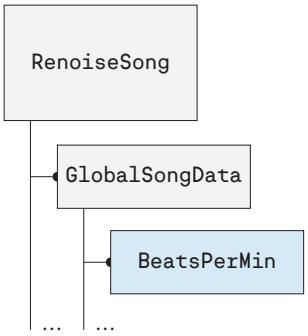


Figure 2.4: The BPM value in the .XML.

With further analysis of the file, it was possible to retrieve the values inside each pattern, as seen in figure 2.5.

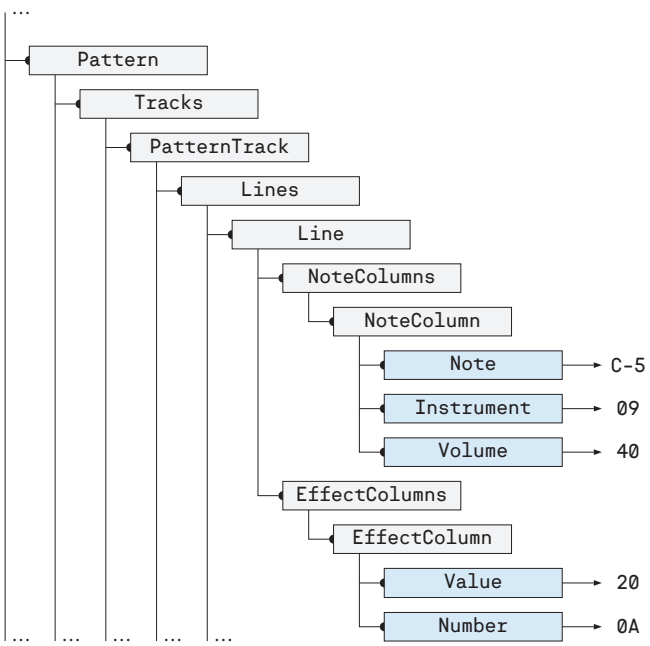


Figure 2.5: The structure of a pattern in the .XML.

Chapter 3

Solution Development

The main purpose of this project in this phase was to create a platform that could load an XRNS file, interpret its messages and playback the audio. This way, depending on the different game events, it would be possible to trigger several behaviors in the track, such as changing tempo, muting channels, changing loop sections, and many more. Following the methodology described in Chapter 2, the proposed architecture focused on four fundamental elements:

- Composition environment used to create the tracker files;
- Development of a parser script to retrieve the information from those files;
- Development of an audio engine prototype which would receive the information obtained with the parser;
- An implementation of the prototype in a real game.

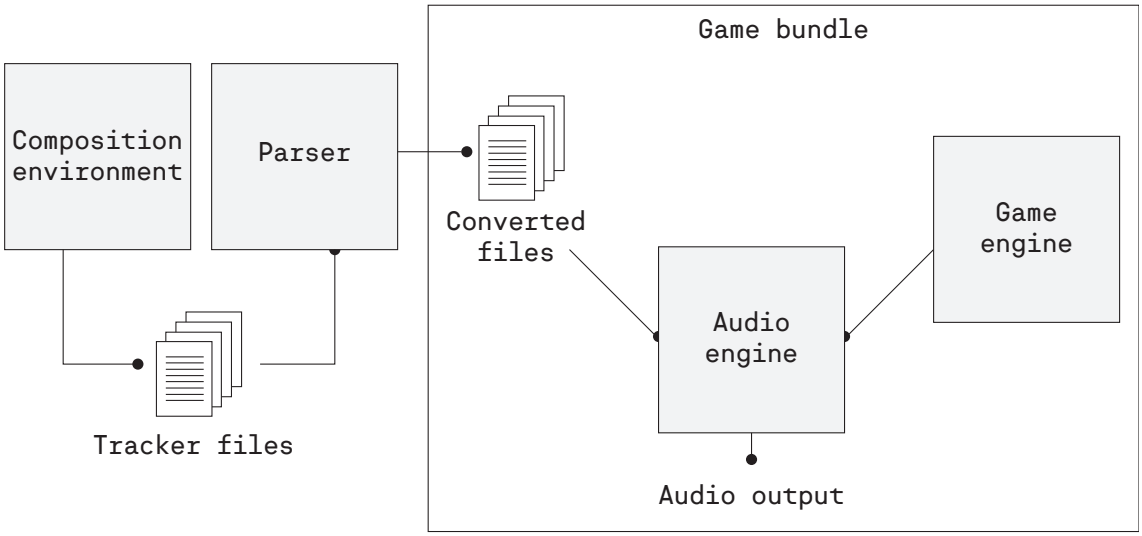


Figure 3.1: Diagram of the proposed architecture.

The diagram presented in figure 3.1 describes the behavior and workflow of this architecture, which would later be validated with the game integration[1].

In order to accomplish this, it was necessary to select the tools and frameworks to use. For the composition environment it was chosen Renoise, as it proved to have the most open and accessible file format during the research. It also features a very comprehensive language to store the music information, which is the XML. With the analysis of the file came the idea of accessing the stored values and extract them to be interpreted later in something that ended up being a music player engine.

The most convenient way to extract the data was to program a parser in a scripting language, in this case Python, for being modular and easy to use/understand. For the audio engine itself, the chosen tool was Pure Data, because it is a powerful data flow environment, capable enough to create a complex music system such as this.

[1] The validation of the proposed architecture is discussed further in Chapter 4.

3.1 Parser

The parser consisted in extracting the most relevant data from the .XML and then export it into a more readable data structure to use in Pure Data. The main focus was to access each pattern information and write it with normalized[2] values in a text file (*module.txt*), in order to have a music score of the module file, as seen in figure 3.2. The chunks of four values demonstrated in the picture represent a typical tracker message in the following order: Note - Instrument - Volume - Effect. Everything was written as whole, with no track or pattern separations, meaning that the first pattern would start at line 1 but the second would start at line 65, the third at 129 and so on.

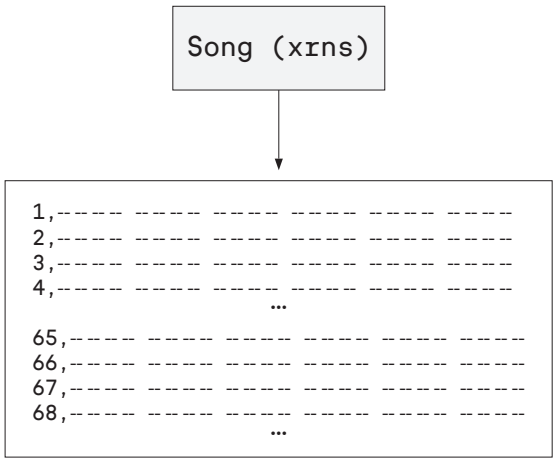


Figure 3.2: Song to text example.

A special command was created specifically to be read in the audio engine, which enables the user to jump to other patterns, in order to create looping sections. This was done by using the two fields (ID and value) of the effects command, where the ID corresponds to the pattern, and the value corresponds to the line.

$$JumpTo = (PatternID - 1) \times 64 + LineNumber \tag{3.1}$$

This command was created using numbers that Renoise does not identify as commands. For example, if the user wants to go to the line 20 (LineNumber = 20) of the second pattern (PatternID = 2), the resulting line number in the text file would be 84. For this particular example the command would be "120", where the first character (1) identifies the pattern, and the rest of the characters (20) identifies the line to jump to. Figure 3.3 demonstrates this command being used in Renoise[3], where the user wants the music to jump from the current line to the first line of the first pattern ("101").

Figure 3.3: Custom *JumpTo* command.

[2]The values that are retrieved with this parser script are used later in Pure Data, described on the Audio Engine section. During the development it was found that it was more convenient to read the data already converted to float numbers instead of converting it in PD.

[3]The custom command used to jump to other lines is specific to the audio engine that was developed as part of this prototype. The command does not mean anything to Renoise.

There are three more text files along with this one. These contain global information which is shared with the song, including a list of all audio samples used in the project, a list of lines that identify the beginning of each pattern and a list with generic information such as the beats per minute for example. In the end the resulting files would be *module.txt*, *module_info.txt*, *module_samples.txt* and *module_patterns*. The procedure is demonstrated in figure 3.4.

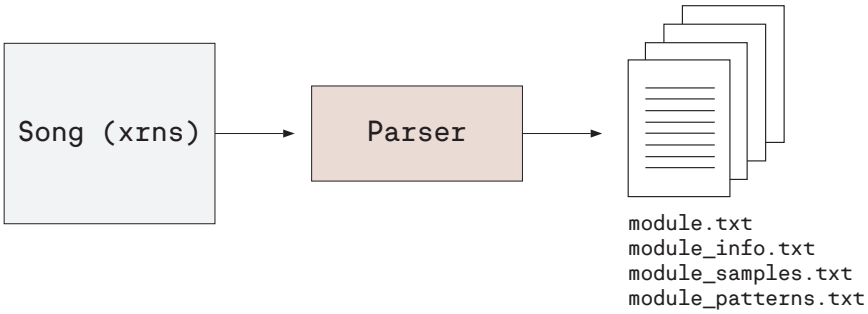


Figure 3.4: The data parsing procedure.

For *module_info.txt* the first step of the script was to retrieve the BPM value of the song and convert it to milliseconds per line. Since a line is equivalent to a sixteenth note (1/16) in a standard pattern (64 lines, 4/4 time signature), the value in milliseconds would be how much does a sixteenth note last in a given BPM value. In order to get a line's duration the following formula was used:

$$Line = (60000/BPM)/4 \tag{3.2}$$

A minute lasts 60000 milliseconds. When dividing a minute by the BPM value the result is the time in milliseconds per beat, which is equivalent to a quarter note's duration. Thus, it is necessary to divide a quarter note's time by 4 to retrieve the value of a sixteenth note. This conversion is important to the audio engine because it needs the duration of each line to play the song with the same tempo used in the Renoise project.

Along with the BPM, a value indicating how many lines does the song have is also saved in this file. Examples of these text files results can be seen in appendix C.

Afterwards, a list of all the samples used in the project is retrieved. The script reads the audio samples folder inside the .XRNS and then extracts them, along with *module_samples.txt*, which contains each sample ID.

Finally, one more text file is exported, the *module_patterns.txt*, which contains a list of all patterns, enabling the user to choose which pattern to play in the audio engine.

3.2 Audio Engine

At this stage of the development there were two main procedures: the first being the text interpreter and the second - the sampler. This sampler was based on a patch from PD's documentation, most specifically the *sampvoice.pd* which is part of the series of examples in *3. Audio Examples*. These patches usually come with most of the Pure Data installations. A patch[4] was programmed to read the previously mentioned text files and perform different tasks with their data.

One of the most important elements is the clock that is running in the background at the milliseconds that were retrieved with the parser (see section 3.1) and stored in *module_info.txt*. This clock is responsible to count values which are used to call their respective line's data in *module.txt*.

For example, the song that was being tested had a BPM value of 154, which results in 97.4 milliseconds after running the parser script. Now this clock will count linearly (1, 2, 3, 4...) with a delay of 97.4 milliseconds between increments, in order to retrieve every line at the song's tempo.

When the program gets a line it splits its data into individual messages (those previously mentioned chunks of four values), and then sends them to the sampler to finally play the audio. This procedure, shown in figure 3.5, was done in order to have the song separated by tracks again.

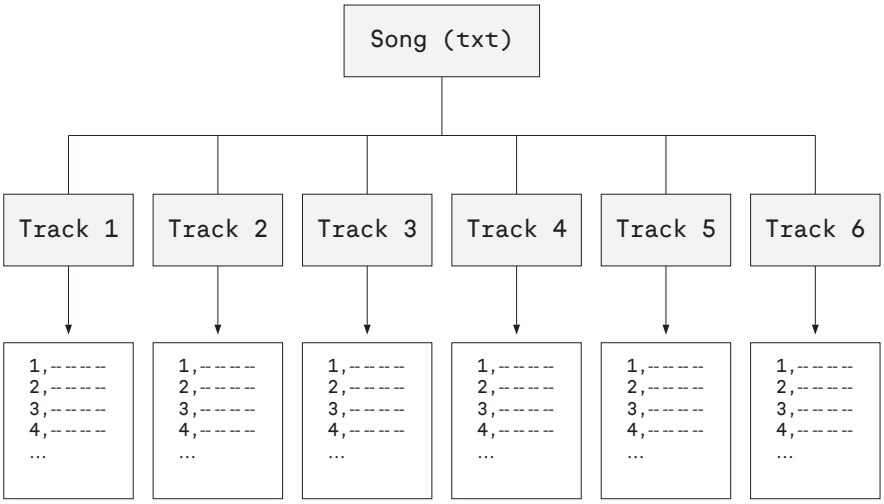


Figure 3.5: The main procedure behind the audio engine.

The patch reads the *module_samples.txt* to find the .WAV files inside the sound folder and load them with an unique identification, the same that is used on each line's instrument number. This way the engine will know which audio sample to play when reading a given message.

Inside the *module_patterns.txt* are the lines which identify the beginning of each pattern. In the audio engine, when inserting a number (the pattern ID), a value (the line) is sent to the main clock to make the music play from that line. The process is similar when a *JumpTo* command is

[4] In Pure Data and Max/MSP the project's window where the user programs his system is often called patch.

found, the patch reads the result of the command (ex: 101 = 1)[5] and will reset the clock to that number.

Having read and interpreted all the text files, the messages retrieved from the *module.txt* are sent to the sampler. Now the patch will read the different numbers of each message and reproduce the audio accordingly.

As mentioned already, this sampler was based on a patch from PD's documentation, most specifically the *sampvoice.pd* which is part of the series of examples in *3. Audio Examples*. These patches usually come with most of the Pure Data installations.

The example uses a message structure to playback audio samples which is very similar to the way a tracker works, with the difference that instead of being the typical Note - Instrument - Volume - Effect, it reads the following values: Pitch in halftones, Amplitude (dB), Duration (msec)[6], Sample number, Start location (msec)[6], Rise time (msec)[6] and Decay time (msec)[6].

It is important to notice that this structure contains seven commands only, although it was mentioned before that the messages were converted from four to eight values. The eighth value was one of the additions, and is used for the custom *JumpTo* command. The other modification to *sampvoice.pd* was the support for stereo, as the original only supported mono.

The value conversions and normalizations done with the parser script (see section 3.1) are described in table 3.1.

Original .XML command	Original .XML value type	Equivalent command	Type conversion
Note	Music notation (ex: C-3)	Pitch in halftones	MIDI value (0 to 127)
Instrument	Hexadecimal (00 to FE)	Sample number	INT (0 to 255) Up to 32 only
Volume	Hexadecimal (00 to 7F)	Amplitude (dB)	1. MIDI (0-127) 2. MIDI to dB (0-100)
Effect	Text & Hexadecimal (ex: ZT 7A)	Not supported* JumpTo (ex: 101)	INT (ex: 1)

*Only the custom JumpTo command is supported in this prototype. Original effect commands are planned for future works.

Table 3.1: Normalizations made in the project.

During the development it was necessary to get back to the parser code and make changes there, as a way to get the messages in PD as ready to play as possible. In order to get parameters such as the duration, it was necessary to include some code that extracts this value from the .XML.

[5] This result is obtained with the formula presented in equation 3.1 from the Parser section.

[6] These do not exist in the tracker format but can be used as result of an effects command conversion.

However, the note length does not exist in a tracker because its duration depends on the number of the succeeding lines. For example, when a note is played, it will only stop when a new note/mute command is found or if the corresponding audio sample has reached its end (if the sample does not have a loop region defined). What does exist in the .XML is the total number of samples (frames) of an audio file. These were fetched and converted to milliseconds at the sample rate of 44.1 KHz, as shown in equation 3.3.

$$SampleDuration = 1000/(44100/TotalSamples) \tag{3.3}$$

The result is the duration of the original audio sample in milliseconds, but this applies only to the base note of the corresponding instrument, which is a value that is also described in the .XML. After this conversion it was necessary to find the duration of the sample after changing its pitch. In order to accomplish this it was necessary to take into account the base note as part of the chromatic scale, as well as its duration. Afterwards the equation 3.4 was used.

$$SampleSizeAfterPitchChange = SampleSize/2^{(Note-BaseNote)/12} \tag{3.4}$$

This calculation provided the sample size depending on its pitch, but it can also be used with any measurement. For example, the duration retrieved in equation 3.3 could be used here instead of the sample size. In order to check if the results are correct, some tests were made in Audacity, which consisted in analysing the size of a sample and change its pitch using the result of the equation. This test is shown in figure 3.6.

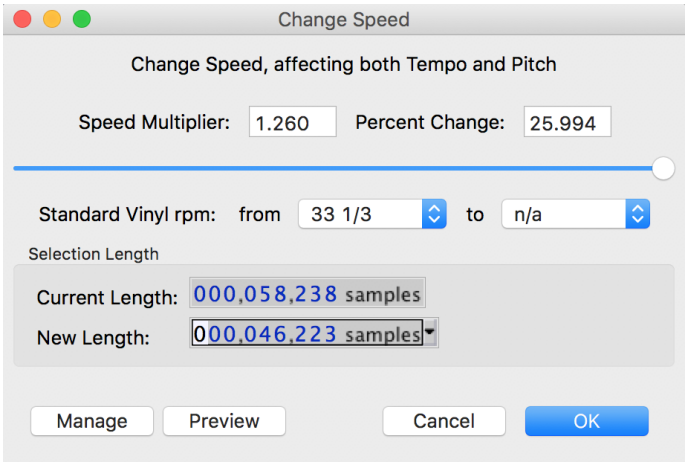


Figure 3.6: Changing pitch in Audacity using the resulting sample size from the equation.

The audio file used in the picture is a C-4 (60 in MIDI). When changing the pitch to an E-4, the difference would be 4 semitones, as 60 - 64 = 4. Using the E-4 note in the formula would provide the respective sample size for that pitch. The two versions of this sample were later analysed with a tuner, as seen in figure 3.7, to check if the pitch shifting was working correctly.

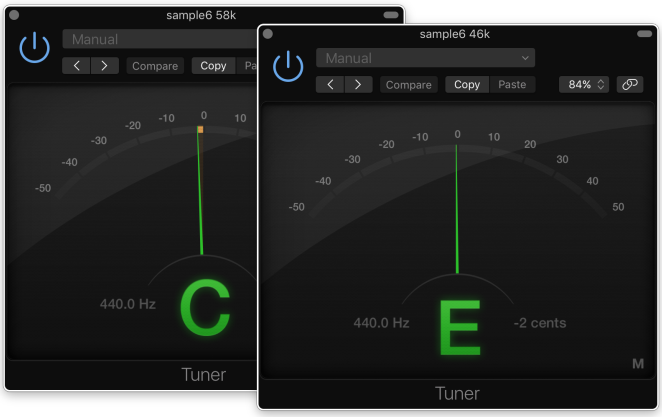


Figure 3.7: Using a tuner to analyse the pitch change (original - left; pitch-shifted - right).

The process of converting and shifting the sample sizes to milliseconds was done to loop regions defined in individual audio samples as well. These loop regions are also described in the .XML file, together with the sample size. Although this was done in the code, the actual sample looping was not implemented in PD[7].

The note duration is written in every message inside the *module.txt* file, together with the rest of the commands. The ones that are not currently being used are Start location (msec), Rise time (msec) and Decay time (msec), thus in order to perform tests with the prototype these three commands' values are forced to zero. It is different to have a field with 0 than having an empty field, to distinguish, every command that does not have a value has a "-" symbol written in the text file. These differences are demonstrated in figure 3.8, showing the sample duration in every message.

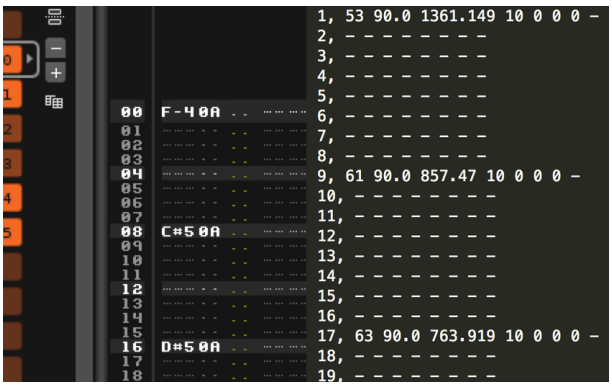


Figure 3.8: Differences between Renoise and text notation for the PD prototype.

However, since the loop function and other associated features were not implemented in the final product, the duration command was deactivated. This way, the engine will always play the full duration of the sample without looping (a technique usually called one-shot sample) and it was possible to remove the unused commands. For more information about these limitations read chapter 6.

[7]Not to be confused with pattern loops, which were implemented.

3.3 Implementation

The implementation of the prototype in Unity was made using local network sockets that enabled a direct communication between the game engine and the music engine. The transmission though is not bi-directional, meaning that messages are only sent from Unity to PD and not the opposite. In short, the music changed depending on the player's actions, but the music couldn't influence the game because there was no communication on that direction. Having this kind of higher interactivity is something that is planned for future iterations[8].

Going back to the architecture's diagram (figure 3.9), this part of the development was centered in the game bundle, where every element works together in real time, as opposed to the parser and Renoise, which were external resources to the model.

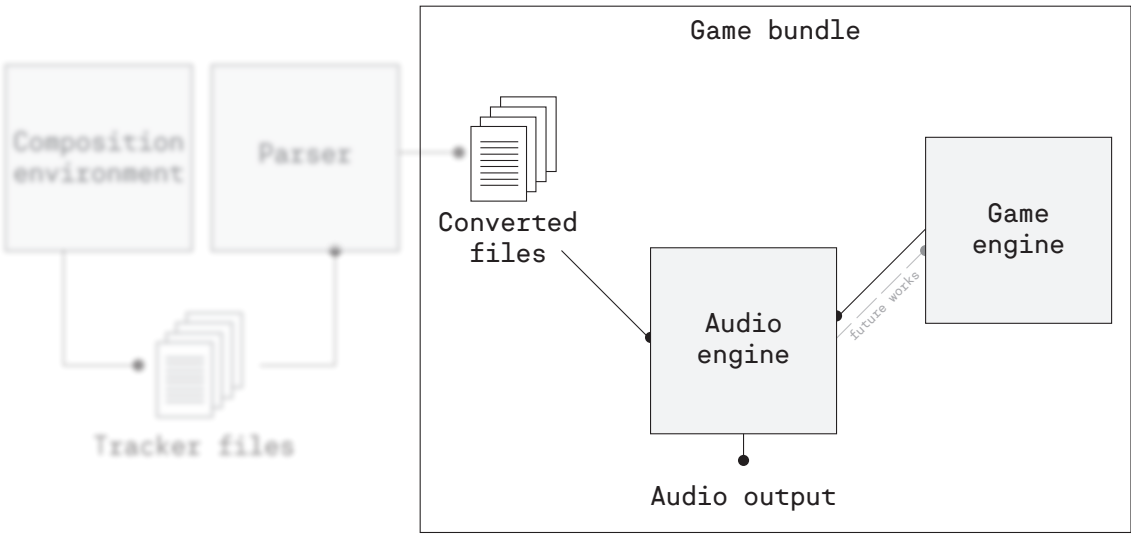


Figure 3.9: The implementation stage, focused on the game bundle part of the architecture.

The PD files and the converted music files were placed in the project's StreamingAssets folder, which is the default place Unity looks into when it is necessary to call this kind of contents. The way this works is by having the patch opened in Pure Data, which reads the converted music and awaits for instructions coming from the game engine. These instructions are basically messages that can trigger/change elements from the music engine's interface, which include the play/stop, BPM value, channel volume, and many others, as long as they have an ID assigned. The messages are sent through functions from the C# scripts.

These functions come with Kalimba, a LibPD[9] library distribution that is used to compile Pure Data patches to be implemented in Unity.

[8] More information about future works is discussed in Chapter 6.

[9] LibPD is an embeddable Pure Data library, often used when it is necessary to compile PD patches into applications, plug-ins and other solutions. There are several distributions made by developers that can be used for different purposes, Kalimba is one specific for mobile platforms.

Attempts to compile the prototype were made, in order to have an integrated environment, without the need of having a PD instance open, which would enable an application build. However, the compilation passed to future works, described with more detail in chapter 6.

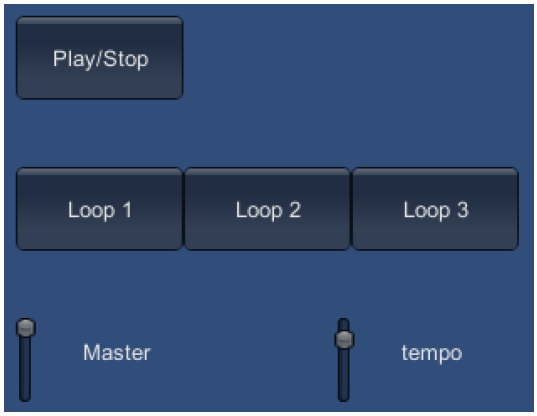


Figure 3.10: A simple interface made in Unity to control different parameters of the music engine.

To test the aforementioned functions, in order to have Unity controlling the developed audio engine, it was necessary to define some receivers on PD's side, and then send the intended message to the respective receiver. Putting it into practice, a simple interface was created in Unity to control some parameters defined in the prototype, as seen in figure 3.10. With the example given, it was possible to start or stop playing the song, change a track's volume (master track in the example), change the tempo, and alternate between loops that were defined in the song using the custom *JumpTo* command, referenced in section 3.2. The controllers on the figure could, instead of buttons, be certain events from a given game, which could occur depending on the player's actions and decisions during gameplay. These parameter controls can be constantly expanded with new features to provide more and better interactivity with this tracker engine.

Below are referenced the messages that were defined for this prototype, which can be used in any Unity project:

Receiver	Message
play	- Start or stop the music (1 or 0)
tempo	- Change the song tempo (bpm)
pattern	- Go to pattern (INT identifying the pattern)
mixer_ch1-32, master	- Change volume of a channel (0. - min, 1. max)
pan_ch1-32, master	- Change panning (0. - left, 0.5 - center, 1. - right)
room_level	- Change the amount of reverb (wet) signal (0. - no reverb, 1. - max level)
room_size	- Change the reverb size (0. no reflections, 1. max size)
reverb_ch1-32	- Set the amount of a channel's signal to send to the reverb effect

Chapter 4

Proof of Concept

After the research, analysis and development stages, the prototype reached a point where some practical tests could be conducted. Having the main features implemented, there were two main keys necessary to validate the proposed method. The first was the process of composing the music in Renoise, export the song and use it in the developed architecture, in order to understand if it was working as expected, as well as to evaluate its capabilities and workflow. The second was to test the music integrated in a real game, which would help getting a clearer view of the entire result.

4.1 The Game

Colleague José Raimundo from the same academic institution was conceptualizing the art and rules for a physical board game entitled *See, Ear, Feel no Evil*. The main ideas from this project were so interesting to the point that it could grow and be adapted to a digital version, which happened some time later. Knowing the people involved in the project made it possible to share ideas related to game events and player choices, which would be fundamental to the music.

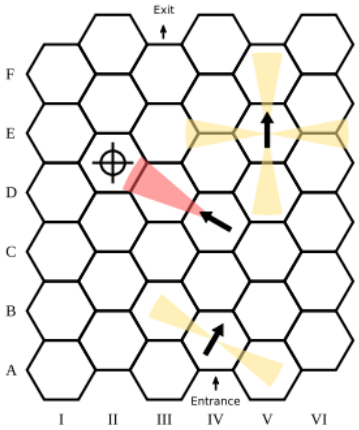


Figure 4.1: Schematic of the game world's map, showing each player's sense radius.

This digital version is a turn-based strategy game, with a *steampunk* theme, where the player can choose one from the three available player classes. Each of these classes has a special capability which is a human sense, more specifically, *sight*, *hearing* and *touch*.

These capabilities have an huge influence during gameplay, because the world map, disposed in an hexagonal grid, is obscured by a very dark fog which can only be dissipated by the radius associated to the class that was chosen, thus reducing the field of view. This effect can be

seen in figure 4.2. The player may encounter enemies to defeat and/or *powerups* to pickup on the level, which can be batteries to charge the door, allowing it to be open, for example. The main objective is to survive and exit the map in safety. For this, a door must be open which spends the available energy. It uses a dice mechanism in order to move in hexagons[1].

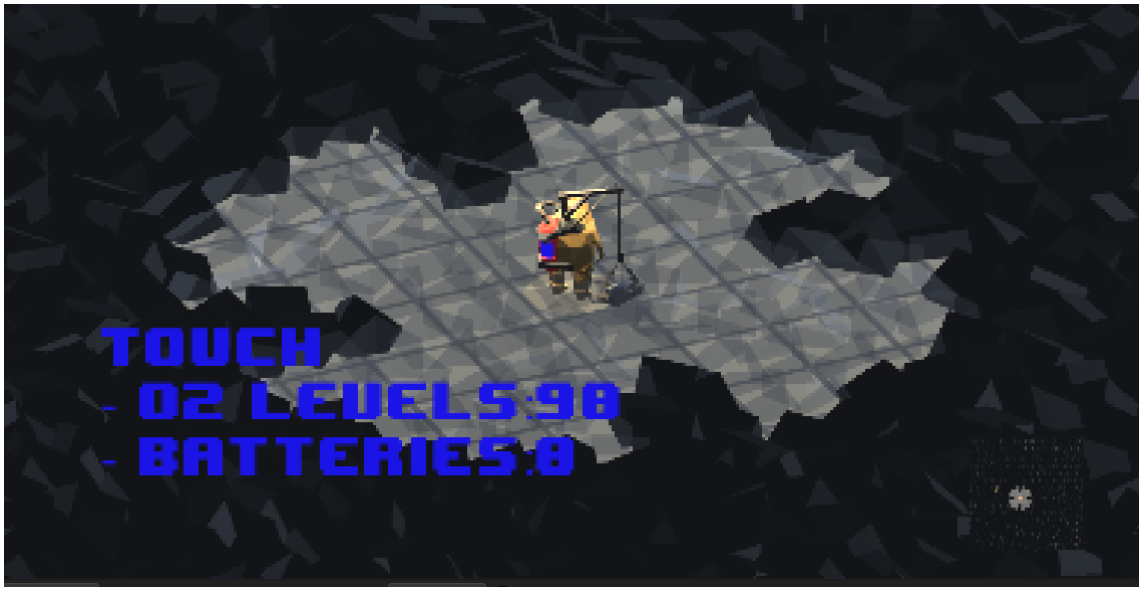


Figure 4.2: A screenshot from one the "See, Ear, Feel no Evil" development stages.

4.2 The Music

The game demo resembles a very deep and dark atmosphere that would suggest a similar effect music-wise, which was the kind of approach used in this project. The composition was based in layers of soundscapes which were constructed using Logic Pro X, and then bounced to audio samples to be used as instruments in Renoise. The idea of having a very limited field of view influenced this approach of using mysterious and eerie sounds, which can help enhance the immersion of the game. The game rules and mechanics have some level of complexity, which allowed the possibility to compose at least two loops for the song.

A few types of musical events were decided during the composition stage, to enable the integration between the song and the player's actions. These would affect parameters, such as the BPM or a loop alternation for example, depending on the respective game event. In short, changes would be made during the game's course, not only to enhance the experience but also to give players the notion of progress. Table 4.1 describes the types that were defined for *See, Ear, Feel no Evil*.

[1]The hexagonal grid changed to squares in the digital version of the game.

Control mapping		
Game Event (Unity)	**Sound Parameter (Pd)**	**Type**
Oxygen	BPM	Tempo
Batteries	Track gain (mixer)	Instrumentation
Enemy	Room size and level (reverb)	Signal Processing
Door	Pattern Change	Horizontal re-sequencing

Table 4.1: Music event types defined for *See, Ear, Feel no Evil*.

All players start the level with a certain amount of oxygen, and during gameplay they gradually lose it. This event is tied with the song's BPM, meaning that as the oxygen level decreases, the music's tempo gets slower. Players can also recover more oxygen which will speed up the music again. During the implementation stage it was decided to do it like this in order to give players the idea of a slumbering effect, caused by the loss of oxygen. It could be done the other way around, since a faster music would resemble panic and an increasing heartbeat, associated with the oxygen loss.

There is also a feature to collect batteries. Whenever someone picks up one, an instrument from the rhythmic section of the music is introduced. This is done by raising the instrument's volume, which starts at zero at the beginning of the level. The interactive changes to the music layers resemble the vertical re-orchestration technique. The instrumentation is organized with the following structure, though only a few of them were mapped in the game:

Rhythm:
1 - Kettledrum
2 - Kettledrum (low)
3 - Tabla (1)
4 - Tabla (2)
5 - Drum ping
6 - Gong
17 - Hi-Hat

Soundscapes/synths:
7 - Dark hit
8 - Lead (fluids)
9 - Lead
10 - Creepy effect
11 - Main synth

Strings:
12 - Double bass
13 - Viola
14 - Violin

Plucked:
15 - Harp
16 - Guitar

Depending on the number of batteries the player has, the music will have more or less rhythmic instruments, with the exception of the hi-hat, which was sent to position 17 because it does not belong to the main loop.

Next are the enemy events. When a player finds an enemy it will produce changes with signal processing, in this case reverb. To give the idea of danger the engine blurs out the music through the spreading of sound.

Finally there is the door event, which starts playing a new music section whenever it is opened, the second loop.

Some mapping of the aforementioned parameters can be seen in figure 4.2, more specifically the oxygen controlling the tempo, and the number of batteries controlling the instruments' volume.

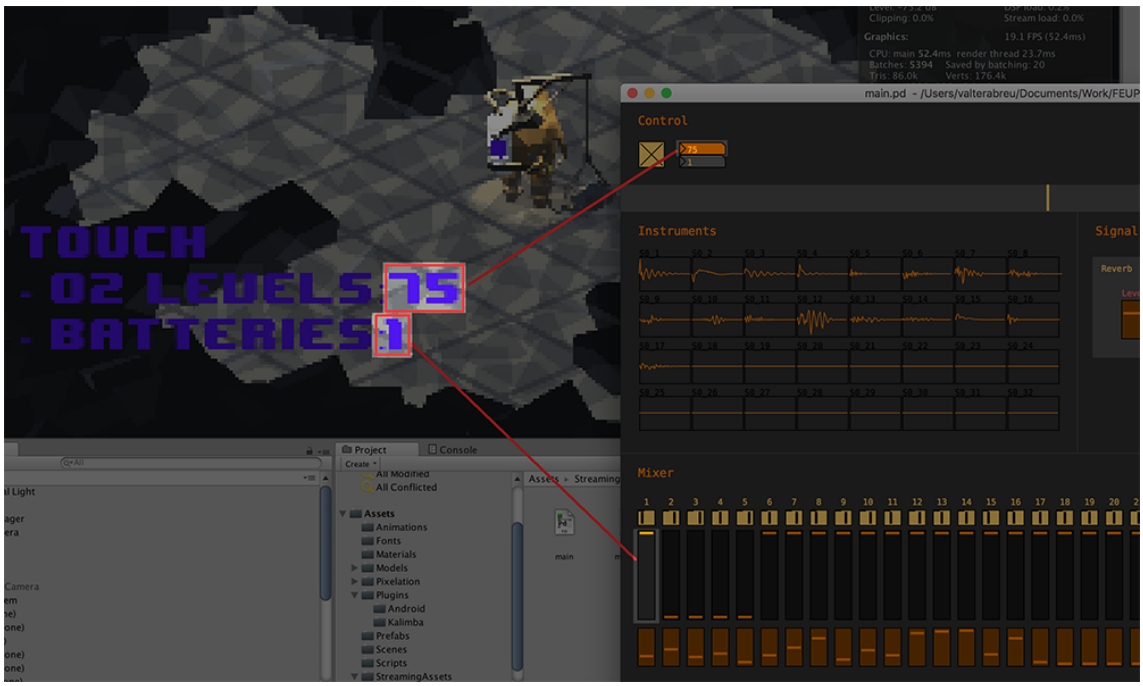


Table 4.2: Mapping game events to music parameters.

Chapter 5

Results

After the implementation tests of the prototype in the game, it was time to analise its performance, in order to evaluate how it would behave and how much resources it would consume. These results were based not only on file sizes, CPU percentage and memory usage parameters, but most importantly on the solution's workflow.

5.1 Worklfow

Tracker music provides different results in terms of articulations, which behave differently from other solutions such as MIDI. The idea of having an independent effects command is very interesting, as one can apply a given effect and easily switch it to another. This can obviously be extended to the game itself, as it could be possible to send musical effects to a selection of notes that are being performed by the tracker. Modern trackers such as Renoise even have the possibility to apply more than one effect per note, resulting in more possibilities for sound variation.

Using a tracker sequencer seems very convenient. It enabled the possibility of composing a song with different sections that were easily triggered in Unity. Besides the existing commands of most trackers, it is possible to add custom commands that can enable new features. An example of this was the creation of the *JumpTo* message, that enabled the fast definition of section loops and it is easily accessible when integrating in a game.

This tool can also be used for sound design approaches. Since the access to several receivers is simplified, the soundtrack can have instruments that are used to receive notes from the game, in addition to the ones that belong to the music. This way a given game event can be programmed to trigger independent notes. For example, repetitive events such as a gun fire or even footsteps could be used to send series of notes to the tracker and play them accordingly, using a musical instrument or a set of realistic samples. This feature is not only for note messages, it applies to several other parameters too because all access is dynamic. Doing this is important as it enhances the coherence between music and sound effects, which will influence the complete user experience.

The idea of bringing a tracker music player to Unity using a data flow programming environment extended the concept to a more dynamic architecture. This means that the proposed solution is modular, as it is possible to change parts of the prototype and also add new features without the need of changing the game's script. Without Pure Data, most of the accomplished tasks would probably need more time to develop and would be less intuitive.

Composing music for games using a tracker software suggests being an ideal middle-ground between musicians and programmers, as during the integration stage the communication between the two profiles was relatively easy to establish due to intuitive identifiers for each parameter, as shown in figure 5.1. The music in the end is pre-programmed by the composer and ready to be controlled by the different game events.

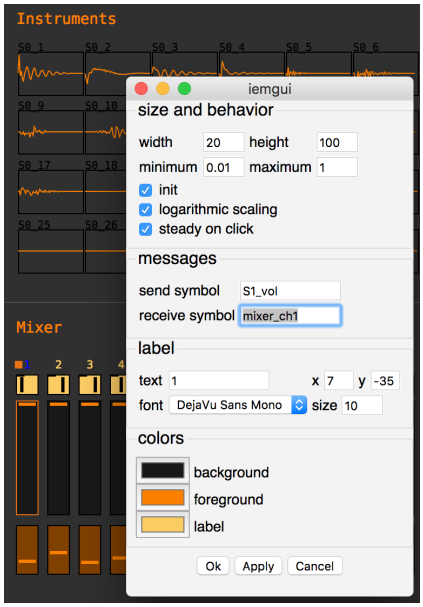


Figure 5.1: Defining a volume slider's identification in PD.

5.2 Performance hit

Using the Task Manager tool while running the game and Pure Data on Windows, it showed results which seem to be very acceptable. There was a residual CPU and memory usage percentage from PD alone, making an average of 3% and 45 MB, respectively.

From this analysis it was also concluded that Unity alone, with this particular game, costs usually between 20 to 23% of the CPU and almost 800 MB of memory, making a total average of 25% when including Pure Data, as seen in figure 5.2.

Name	CPU	Memory	Disk	Network	GPU
Unity Editor (8)	25,5%	783,7 MB	0,2 MB/s	0 Mbps	37,7%
Console Window Host	0%	4,9 MB	0 MB/s	0 Mbps	0%
Console Window Host	0%	4,9 MB	0 MB/s	0 Mbps	0%
Node.js: Server-side JavaScript	0%	40,5 MB	0 MB/s	0 Mbps	0%
Pure Data Application (32 bit)	3,0%	43,6 MB	0 MB/s	0 Mbps	0%
Unity Editor	22,5%	647,0 MB	0,2 MB/s	0 Mbps	37,7%
Unity Shader Compiler	0%	15,8 MB	0 MB/s	0 Mbps	0%
UnityCrashHandler64	0%	0,9 MB	0 MB/s	0 Mbps	0%
UnityHelper	0%	26,0 MB	0 MB/s	0 Mbps	0%

Figure 5.2: CPU and Memory usage analysis.

From the aforementioned 3% it needs to be subtracted some considerable amount of percentage, as part of this value is related to graphical interface processes which would not be present in

a compiled plug-in. Since PD was running in stand-alone, the task manager showed almost 1.5% of CPU and about 60 MB of memory usage, as seen in figure 5.3.

Name	CPU	Memory	Disk	Network	GPU
nwjs (32 bit) (4)	1,3%	59,8 MB	0 MB/s	0 Mbps	0,5%
nwjs (32 bit)	0%	1,0 MB	0 MB/s	0 Mbps	0%
nwjs (32 bit)	0,2%	14,7 MB	0 MB/s	0 Mbps	0,5%
nwjs (32 bit)	0,9%	30,5 MB	0 MB/s	0 Mbps	0%
nwjs (32 bit)	0,3%	13,7 MB	0 MB/s	0 Mbps	0%

Figure 5.3: PD's graphical interface hit.

Further analysis suggested that the communication between Unity and PD via sockets had some performance impact. By temporarily removing the Kalimba functions that establish the connection between the two applications, the CPU usage percentage dropped about 3%, resulting in values between 17 to 19% from Unity alone. These results are shown in figure 5.4.

Name	CPU	Memory	Disk	Network	GPU
Unity Editor (7)	19,6%	722,7 MB	0 MB/s	0 Mbps	32,5%
Console Window Host	0%	4,9 MB	0 MB/s	0 Mbps	0%
Console Window Host	0%	4,9 MB	0 MB/s	0 Mbps	0%
Node.js: Server-side JavaScript	0%	40,0 MB	0 MB/s	0 Mbps	0%
Unity Editor	19,6%	631,1 MB	0 MB/s	0 Mbps	32,5%
Unity Shader Compiler	0%	14,9 MB	0 MB/s	0 Mbps	0%
UnityCrashHandler64	0%	0,9 MB	0 MB/s	0 Mbps	0%
UnityHelper	0%	26,0 MB	0 MB/s	0 Mbps	0%

Figure 5.4: Unity's CPU and Memory usage after the Kalimba removal.

While the results could have a comparison with other solutions, they suggest that this architecture would be beneficial, when integrated as a plug-in in modern game engines. However, these performance impacts do not seem to affect the game's experience in this context.

5.3 Storage Footprint

The original song that was composed for testing contains two different loops, the main loop lasts for approximately 2:10 minutes, and the second loop lasts for about a minute. Most of the audio samples were in stereo .WAV format in 16 bit at 44.1 KHz sample rate, other sounds were reduced to mono at 22 KHz to compress their size. In the end, the total size of the .XRNS project occupies 5,7 MB of disk space. After the parsing process, the total bundle including text files ended up with a 7,9 MB size.

In order to understand how much space saving was earned, the complete song was rendered to .WAV format (16bit, 44.1KHz to keep the same audio quality) resulting in a file with 32,2 MB, which is about 5 times larger than a Renoise file, and 4 times larger than the exported bundle. When converting the rendered audio to OGG Vorbis at 320kbps, the file reduces to 6,1 MB, which is much more approximate to the original tracker file's size, although it can compromise audio quality due to compression.

Now, a point that is important here is that in rendered audio, if the duration increases, the file size also increases. For example, if the duration was 6 minutes instead of 3, the .WAV file would have 64,4 MB instead of 32,2, as well as the .OGG which would have 12,2 MB instead of 6,1. The same wouldn't happen with the tracker, if the song kept the 17 samples and the duration increased, it would end up with some extra KBs of text, keeping almost the same size. Results from this analysis can be seen in table 5.1, taken from Finder in macOs.

▼ General: Kind: Renoise Song Size: 5 704 240 bytes (5,7 MB on disk)	▼ General: Kind: Waveform audio Size: 32 208 940 bytes (32,2 MB on disk)
▼ General: Kind: Folder Size: 7 868 362 bytes (7,9 MB on disk)	▼ More Info: Duration: 03:03 Audio channels: 2 Sample rate: 44 100 Bits per sample: 16
▼ General: Kind: Ogg Vorbis File Size: 6 050 767 bytes (6,1 MB on disk)	

Table 5.1: File sizes analysis.

Chapter 6

Conclusions

6.1 Summary

Conclusions show that reusing trackers in modern game development apparently can be an advantage in several aspects. It can make the integration stage easier in terms of code writing, since a programmer does not need to assemble all the different music sections and behaviors through scripting, much like the composer, who does not need to program those behaviors in a game engine, as his song was already organized in that sense, at the composing stage.

It is suitable as a tool for adaptive music, in the sense that individual instruments, panning, and other musical features are accessible to be controlled by the game. A music file may contain several variations and song sections with their respective identifications, as well as their loop settings. This also results in relatively simple structures, since one file only contains everything that is necessary, as opposed to other solutions where sets of files can be quite extensive.

However, trackers can have their disadvantages. The musical language of a tracker is different from the ones which a composer may be used to. For example, in a music sheet or in a DAW, the composing process is made with an horizontal notion of time passing, as opposed to the vertical display used in a tracker. The input method can also be intriguing and difficult to learn, as the idea of using different commands to produce a note event is different from the usual music notation.

By the end of the solution development and results analysis, it was possible to answer the following research questions that were present in the Introduction:

- **RQ1**. *Can trackers be used for adaptive music generation in modern game development?*

  With the development of a prototype that enabled the playback of tracker music files in a modern game engine, it was possible to verify that this method can still be used today. Not only the original concept of this kind of software is valid, but it can also be expanded to be used in a more dynamic way. Even with limited features, the fact that the prototype was developed in a data flow environment helped building a very functional system that could be used in modern game development.

- **RQ2**. *Are trackers a valuable strategy for adaptive music generation?*

  Since the very beginning, trackers have been used to create adaptive music pieces for video-games. The limitations were merely technological since hardware, memory and disk storage were very limited at the time, compromising the audio quality and processing capabilities. Bringing this concept to the present, trackers can be adapted to today's audio standard qualities, which could result in a potential competitor against other solutions. It proved to be feasible in this research, during the composing and implementation of the music in a game, as it enabled many musical conditions, variations and parameter controlling.

- **RQ3**. *What benefits does using a tracker provide in terms of resource usage?*

  Tracker music can be very lightweight in terms of file sizes, as they mostly carry textual information. This can obviously be variable as it also depends on the quality and size of the audio samples used in the song. After validating this method by using it with the developed prototype, it was concluded that it performed with good results, showing no apparent signs of latency or CPU straining. Further studies will be conducted in future works, in order to check if these results in terms of resource usage are the same or similar when using a real integrated environment.

Most of the main goals for this research were achieved, some were surpassed, others were tested but will need more work and research in the future. Below are revisited the objectives presented in the Introduction:

- *Propose an audio platform capable of playing module files in an interactive way.*

  This was accomplished with the development of an architecture that consists in a tracker music file decoder (the parser), and a tracker music player (the audio engine developed in PD). The result was more than positive, offering an interesting and interactive manipulation of the song, as well as a very modular approach to feature implementation.

- *Enable the integration of this model in a modern game engine. For testing the tracker music player, original songs will be composed in order to explore different musical strategies.*

  The model was integrated in a game, which was explained in the Proof of Concept chapter. This enabled the audio engine to play the music in various forms, as well as to decide what parameters should be controlled. For testing, only one song was possible to compose due to time limitations, though it had enough elements to explore the different adaptive music techniques. This can obviously be used with other tracks as well.

- *Evaluate the benefits of using module files in terms of resource usage and understand its limitations and advantages. This prototype should run in mobile platforms.*

  Several tests were made to the prototype's integration showing positive results, which can be seen in the Results chapter. The model does not run in mobile though, as there were problems compiling for iOS. For Android it was possible to build a testing application with kalimba, but there were serious latency problems which could not be surpassed.

6.2 Limitations

Although the prototype covered some important aspects of the tracker music sequencers, there were several other features that could not be part of the development. These limitations were defined at the very beginning of the development process, as some tasks need more time than others and thus, only the fundamental work is present in this prototype.

Current limitations are shown in the list below:

- Currently only 64 lines per pattern are supported;
- Only the 4/4 time signature can be used;
- The prototype reads a sample's loop region, but the actual looping was not implemented;
- Currently only one sample per instrument is possible;
- Channel groups cannot be used, only normal channels like in traditional trackers and only one value per column is possible;
- The number of possible channels is limited to 32;
- Glissando, Note re-triggering, Arpeggiator, Note reversing, and many more classic and new effects are not supported;
- Reverb is the only signal processing option supported.

6.3 Future Work

This project was created with a vision of being greatly expanded. Future implementations will include working on the previously mentioned limitations, in order to make better use of Renoise as a music composition environment for video-games.

These are the main goals planned for future works:

- The possibility of choosing how many lines the user wants to use in each pattern;
- Change a pattern's time signature;
- Sample loops, in order to use sustaining notes until a new command stops it;
- Multi-sampled instruments;
- Channel grouping and multi-valued note/effect commands to enable higher polyphony and versatility soundwise;
- Larger number of samples;

- Complete effects commands coverage;
- More signal processing possibilities.

Another feature would be to integrate the parser in the audio engine itself, in order to eliminate the process of a user converting the track to text. This way the engine would read the Renoise files directly and playback the audio.

To provide more interactivity, bi-directional communication between the audio engine and game engine is a plan for future iterations. This way, not only the game can make changes to the music, but the music could also make changes in the game.

Finally, one of the most fundamental improvements for this project is the complete integration in the game engine. Currently the prototype is completely external to the game engine, as it is only communicating via sockets, but in future work it would be ideal to have this audio engine as a plug-in to use in Unity, and possibly more game engines.

Appendix A

Audio engine prototype

- Prototype development versions
- PD patches

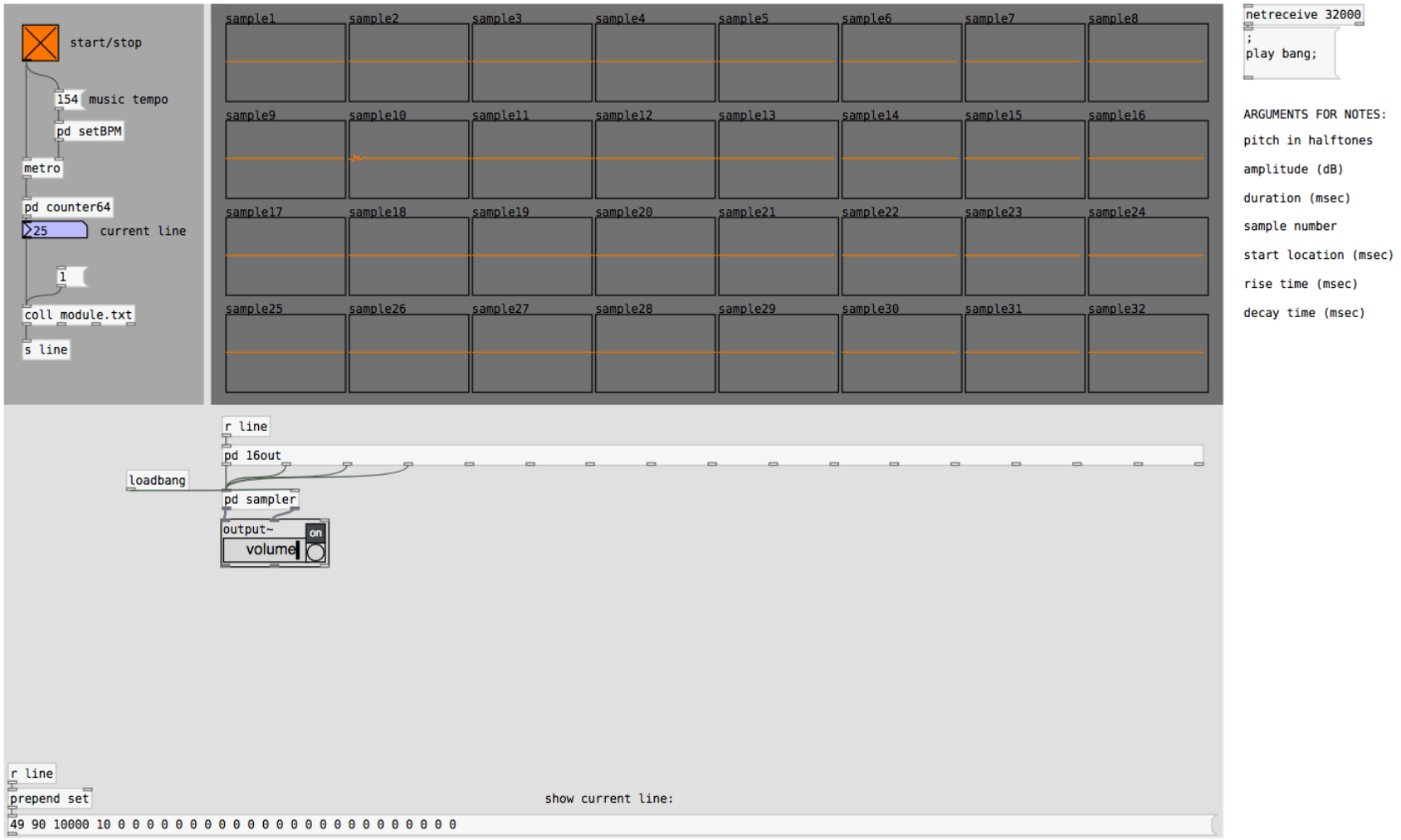


Figure A.1: Early version of the music engine prototype.

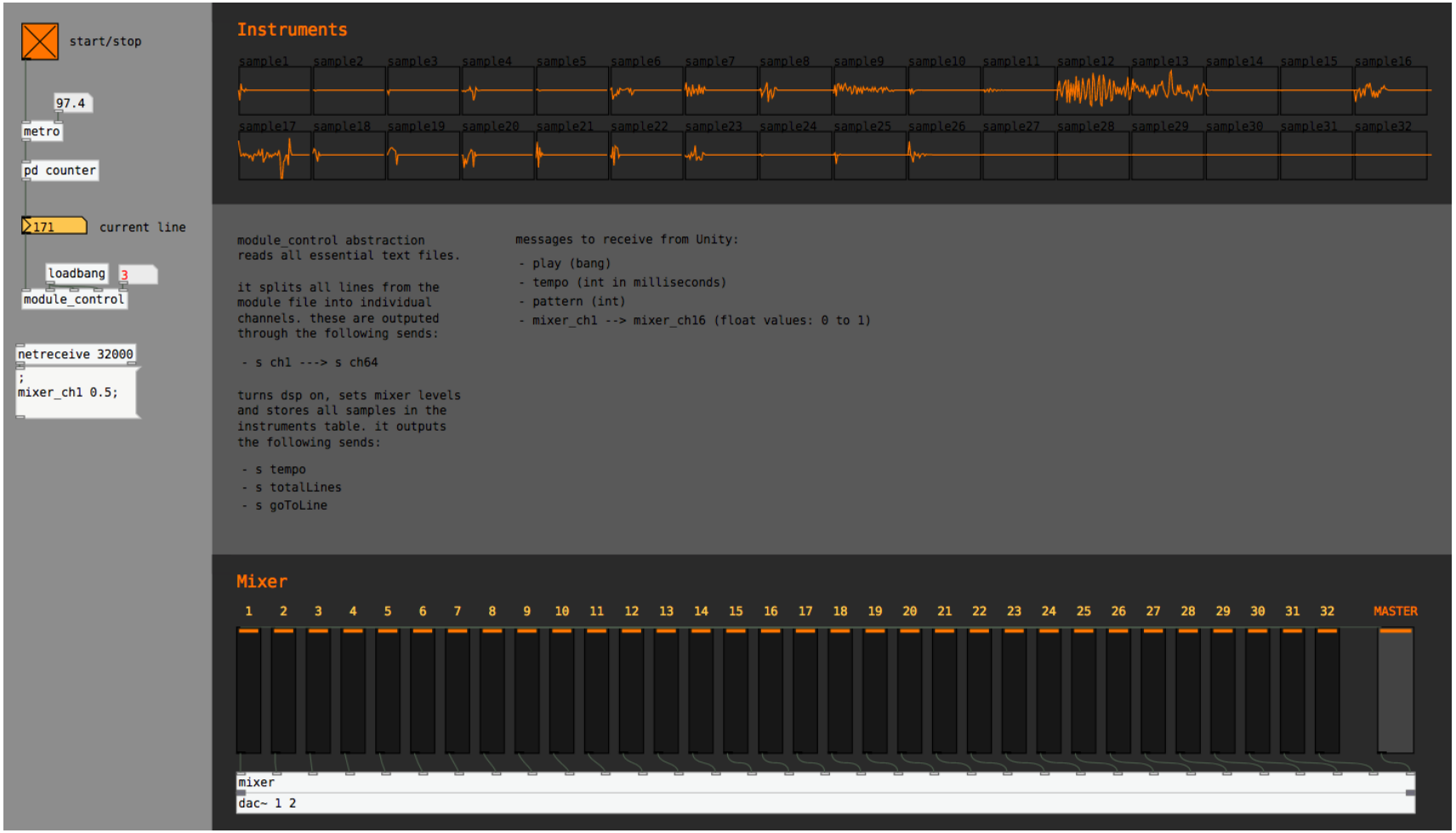


Figure A.2: Development upgrade including interface changes and a mixer.

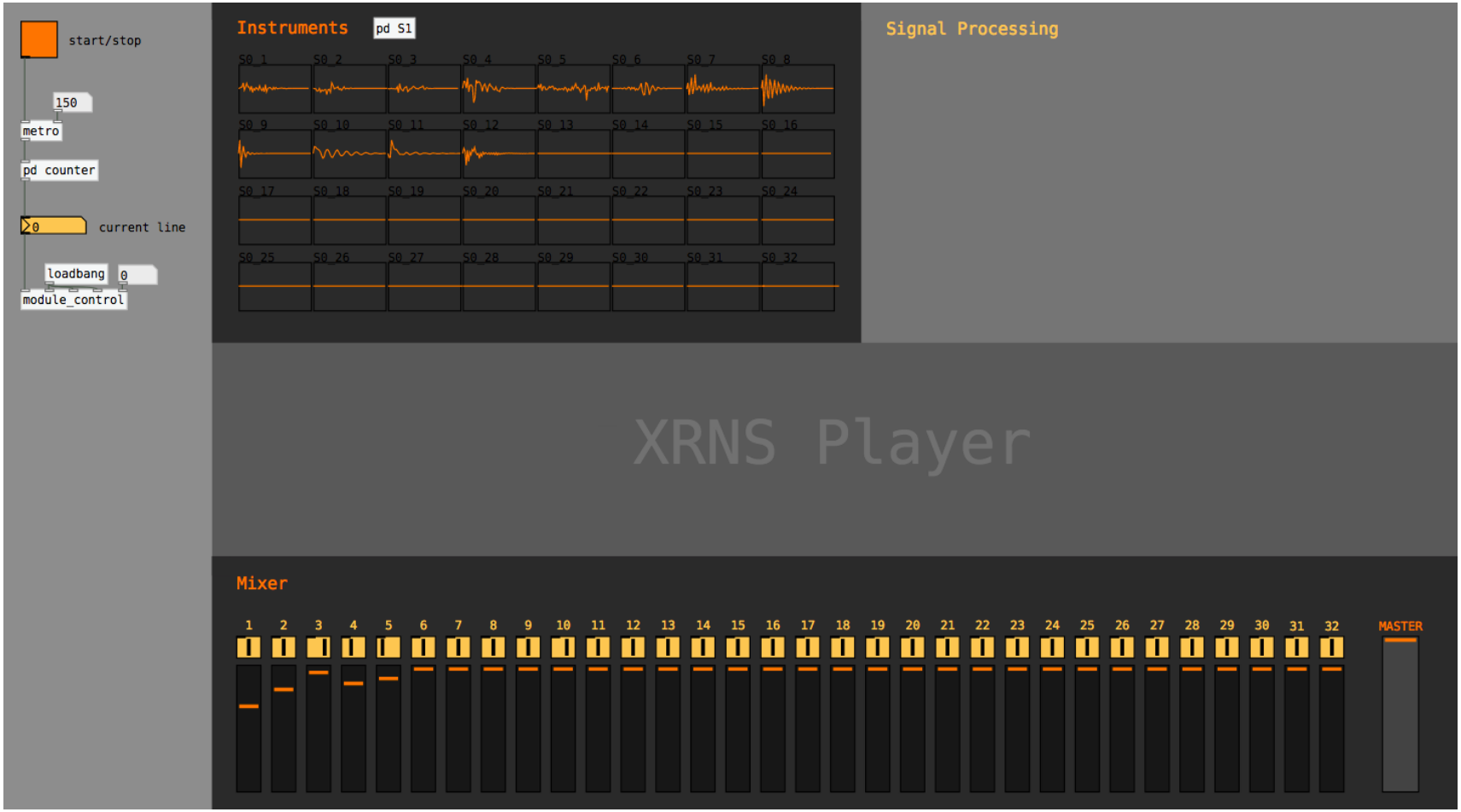


Figure A.3: Development upgrade including stereo and panning features.

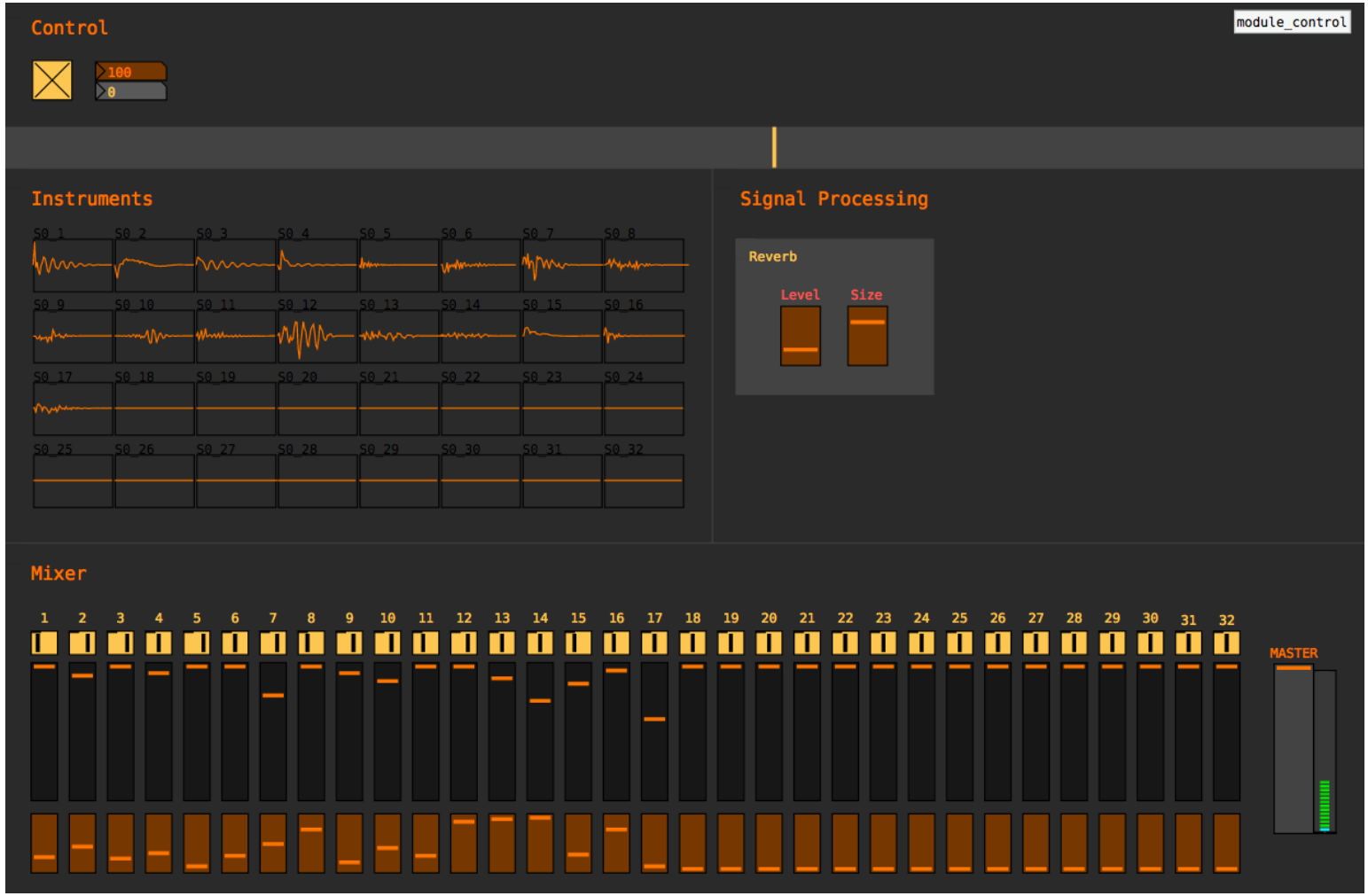


Figure A.4: Final version with signal processing features.

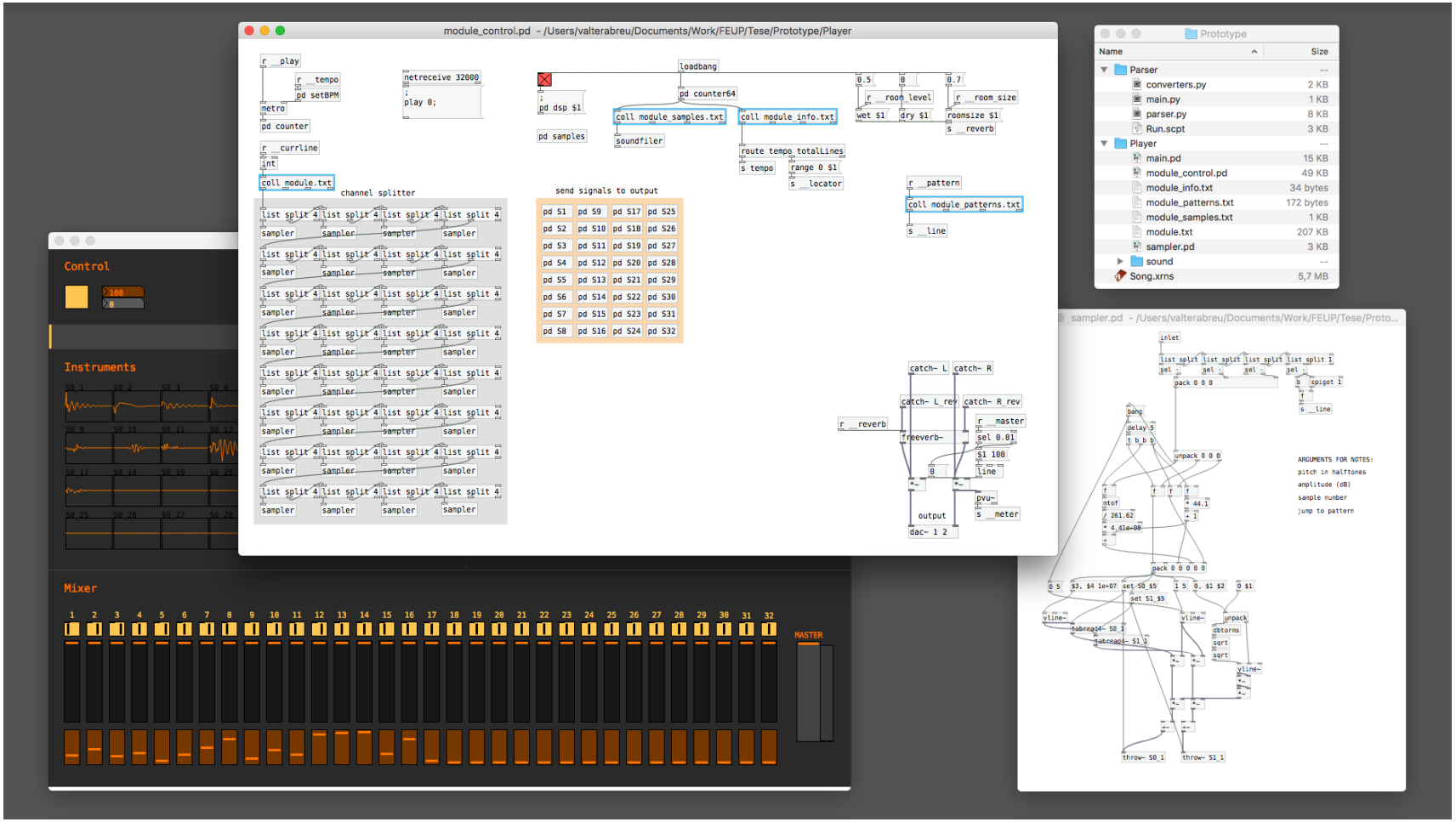


Figure A.5: The project's patch and files.

Appendix B

Source code excerpts

- XML file
- Python script

B.1 XML file excerpts

The very first lines of the .XML document describe the Global Song Data, which includes the BPM used in the project, as seen in listing B.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<RenoiseSong doc_version="63">
  <GlobalSongData>
    <BeatsPerMin>154</BeatsPerMin>
    ...
```

Listing B.1: Renoise's BPM setting (XML)

One of the most relevant sections of the entire document is the one that contains the information of each pattern. The lines that contain any values (note, instrument, volume or effect) are described here, as shown in listing B.2.

```
<Pattern>
...
  <PatternTrack type="PatternTrack">
    ...
    <Lines>
      <Line index="3">
        <NoteColumns>
          <NoteColumn>
            <Note>C-5</Note>
            <Instrument>09</Instrument>
            <Volume>40</Volume>
          </NoteColumn>
        </NoteColumns>
        <EffectColumns>
          <EffectColumn>
            <Value>20</Value>
            <Number>0O</Number>
          </EffectColumn>
        </EffectColumns>
      </Line>
    </Lines>
    ...
  </PatternTrack>
  ...
</Pattern>
```

Listing B.2: Renoise's line structure (XML)

A very detailed information related to the audio samples used in the song is described in the Sample Generator section, containing important data such as panning, volume, loop zone and many more elements. This can be seen in listing B.3.

```
...
<SampleGenerator>
  <Samples>
    <Sample>
      <SelectedPresetName>Init</SelectedPresetName>
      <SelectedPresetLibrary>Bundled Content</SelectedPresetLibrary>
      <SelectedPresetIsModified>false</SelectedPresetIsModified>
      <Name>TYMP2.WAV</Name>
      <Volume>1.0</Volume>
      <Panning>0.5</Panning>
      <Transpose>0</Transpose>
      <Finetune>0</Finetune>
      <BeatSyncIsActive>false</BeatSyncIsActive>
      <BeatSyncLines>16</BeatSyncLines>
      <OneShotTrigger>false</OneShotTrigger>
      <NewNoteAction>NoteOff</NewNoteAction>
      <Oversample>false</Oversample>
      <InterpolationMode>Cubic</InterpolationMode>
      <AutoSeek>false</AutoSeek>
      <AutoFade>false</AutoFade>
      <LoopMode>Off</LoopMode>
      <LoopRelease>false</LoopRelease>
      <LoopStart>0</LoopStart>
      <LoopEnd>1</LoopEnd>
      ...
```

Listing B.3: Renoise's sample information (XML)

B.2 Python script excerpts

In listing B.4 is presented the script's main file. This imports functions from the parser.py file that are responsible for extracting the values from the .XML. It also instantiates the necessary text files to write the data in.

The getSongData() function returns a data structure (a *dictionary*) that contains the complete music score. With this, it is possible to select a specific line from a given pattern and from a specific track to print it. Thus, several *for loops* were used in order to print the entire song.

More functions were defined to get useful information from the song. These are getTempo(), getTotalLines(), getInstruments() and getPatternIds().

```
1  def main():
2    import parser
3    module = open('../Player/module.txt', 'w+')
4    module_info = open('../Player/module_info.txt', 'w+')
5    module_samples = open('../Player/module_samples.txt', 'w+')
6    module_patterns = open('../Player/module_patterns.txt', 'w+')
7    parser.start()
8    song = parser.getSongData()
9
10   # Example for printing:
11   # print(song['Pattern 0']['Track 1'])
12
13   #### Write the track ####
14   counter = 0
15   for pattern in song:
16     for line in range(64):
17       counter += 1
18       module.write(str(counter) + ', ')
19       for track in song[pattern]:
20         for i in range(8):
21           module.write(str(song[pattern][track][line][i]) + ' ')
22       module.write(';\n')
23
24   #### Write track global info ####
25   module_info.write('1, tempo ' + parser.getTempo() + ';\n')
26   module_info.write('2, totalLines ' + parser.getTotalLines() + ';\n')
27
28   #### Write samples list ####
29   for i in parser.getInstruments():
30     module_samples.write(i + '\n')
31
32   #### Write patterns info ####
33   counter = 0
34   for k in parser.getPatternIds():
35     counter += 1
36     module_patterns.write(str(counter) + ', ' + str(k) + ';\n')
```

```
37
38    module.close()
39    module_info.close()
40    module_samples.close()
41    module_patterns.close()
42
43  if __name__ == '__main__':
44    main()
```

Listing B.4: The main.py file.

Most of the code is done in the parser.py file. In this file were defined the functions that are responsible for fetching the patterns, tracks, lines, values, tempo, sample information and more elements. In listing B.5 is shown an excerpt of the value extraction procedure for each line inside a pattern. This process is running inside a *for loop* and stores the values in an auxiliary array (*list*) called "item", which later in the code is stored in the song data structure.

This file is using some special functions such as tagHasValue() and getValueFromTag(). These were defined in a file called converters.py and are responsible to check for values, fetch them and convert them to the desired data type.

```
1  ...
2  # Note/pitch command:
3  if tagHasValue(k, 'Note'):
4    note = getValueFromTag(k, 'Note', 1)
5    item[0] = note
6
7  # Instrument/sample command:
8  elif tagHasValue(k, 'Instrument'):
9    inst = getValueFromTag(k, 'Instrument', 2)
10   item[3] = inst
11
12 # Volume/amplitude/velocity command:
13 elif tagHasValue(k, 'Volume'):
14   vol = getValueFromTag(k, 'Volume', 2)
15
16 # Tags from FX column:
17 elif tagHasValue(k, 'Value'):
18   fxValue = getValueFromTag(k, 'Value')
19
20 elif tagHasValue(k, 'Number'):
21   fxNumber = getValueFromTag(k, 'Number')
22 ...
```

Listing B.5: A parser.py excerpt showing the line values extraction process.

One of the most used functions used in the parser.py is defined in the converters.py, shown in listing B.6, and is responsible to check if a given tag exists (it exists when it contains values). Python's built-in function *find* returns the position of the inputted value if it exists in the information that is being searched, if not it returns a -1 value. Using this, the custom function tagHasValue() returns a True or False when looking for a giver tag. It also works for closed tags, if a True argument is used.

```
1  #Check if a tag has values
2  def tagHasValue(targetText, tagName, isClosed=False):
3    if isClosed:
4      tag = '</' + tagName + '>'
5    else:
6      tag = '<' + tagName + '>'
7
8    if targetText.find(tag) > -1:
9      return True
10   else:
11     return False
```

Listing B.6: Value check function from converters.py.

A function was defined to get a value from a given tag and it was often called in the script after a tagHasValue() verification. If the value type argument is inputted, the function will convert the received values to an *int* value, using some special functions that were defined in the converters.py. It uses Python's built-in *split* function, in order to get information in between text. The procedure is shown in listing B.7.

```
1  #Get value from tag
2  def getValueFromTag(var, tag, valueType=0):
3    ### valueType: Default for STR, 1 for MIDI, 2 for HEX, 3 for INT
4    openTag = '<' + tag + '>'
5    closedTag = '</' + tag + '>'
6    value = var.split(openTag)[1].split(closedTag)[0]
7
8    if valueType == 1:
9      return noteToMIDI(value)
10   elif valueType == 2:
11     return int(value, 16)
12   elif valueType == 3:
13     return int(value)
14   else:
15     return value
```

Listing B.7: Function to get values from a tag and/or convert them.

On listing B.8 is shown one of the defined converter functions used in the previous examples, the noteToMIDI(). It takes a note as argument ("C-4" for example), and returns the equivalent MIDI *int* value. This is done by creating a small data structure that stores all values from 0 to 127 in their corresponding *dict* keys, like "'C-3': 60" for example. If the key exists the function returns its value, if not it returns zero.

```
1  #Note to MIDI values
2  def noteToMIDI(note):
3    notes = ["C-", "C#", "D-", "D#", "E-", "F-", "F#", "G-", "G#", "A-", "A#", "B-"]
4    counter = -1
5    midiData = {}
6
7    for octave in range(12):
8      for i in range(128):
9        if i % 12 == 0:
10         counter += 1
11         if counter > 10:
12           counter = 0
13         key = notes[octave] + counter
14         midiData[key] = i + octave
15
16   if note in midiData.keys():
17     return midiData[note]
18   else:
19     return 0
```

Listing B.8: A converter function from converters.py.

Appendix C

Tracker music score example

- Renoise project
- Exported text files

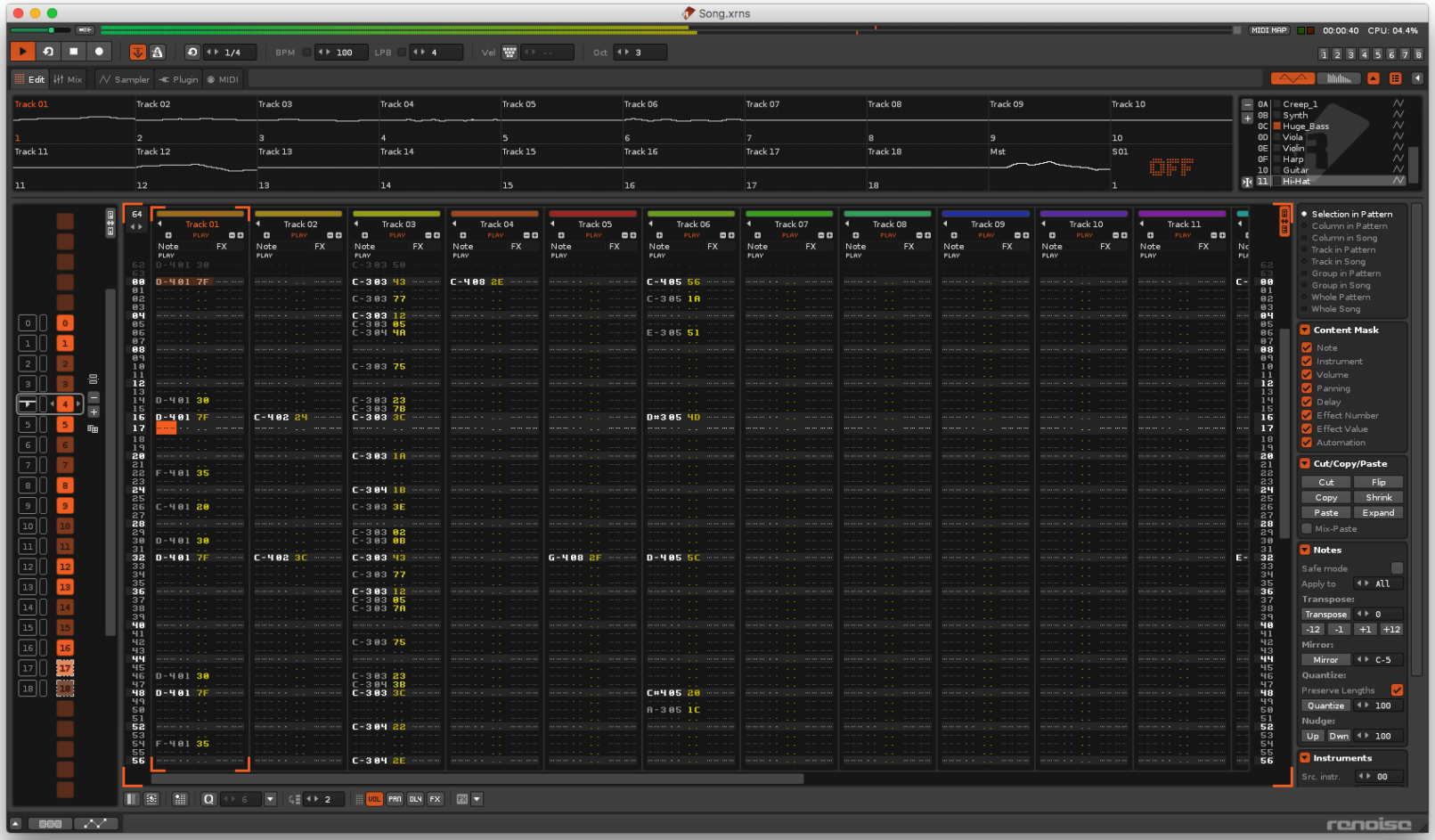


Figure C.1: Renoise song project.

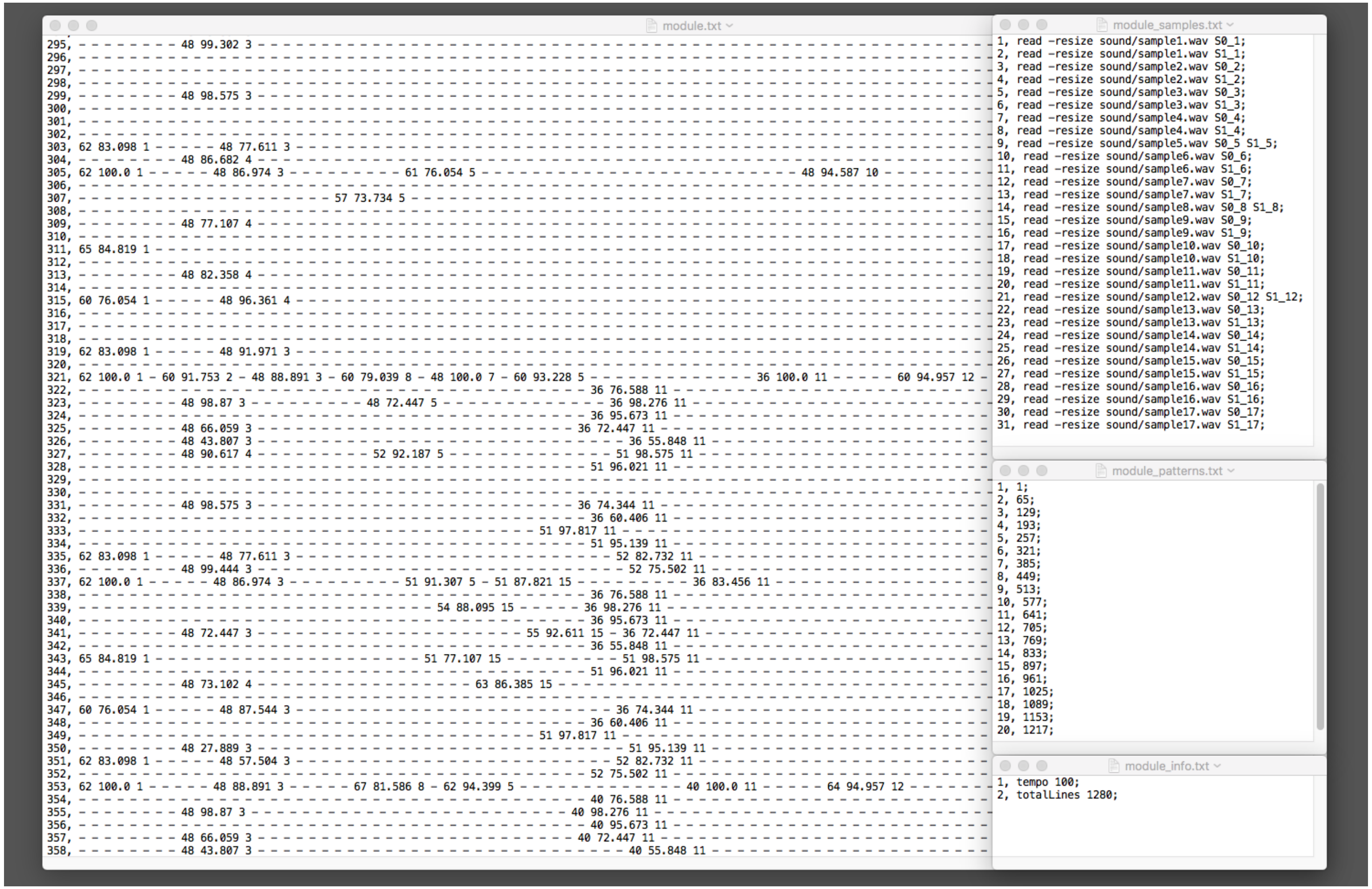


Figure C.2: The music converted to text files.

References

[Andersen, 2011] Andersen, M. S. (2011). "Limbo" – Exclusive Interview with Martin Stig Andersen.

[Brandon, 2005] Brandon, A. (2005). *Audio for Games: Planning, Process, and Production*. New Riders Games.

[Collins, 2008] Collins, K. (2008). *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. MIT Press.

[Gungormusler et al., 2015] Gungormusler, A., Paterson-Paulberg, N., and Haahr, M. (2015). barelymusician: An adaptive music engine for video games. In *Audio Engineering Society Conference: 56th International Conference: Audio for Games*.

[Maher, 2008] Maher, D. (2008). Brian Eno Makes Interactive Music for Spore | Pitchfork.

[Morasky, 2011] Morasky, M. (2011). Portal 2's dynamic music - an interview with composer Mike Morasky, and five tracks to listen to now! | GamesRadar+.

[Nash, 2014] Nash, C. (2014). Manhattan: End-User Programming for Music. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 221–226.

[Phillips, 2014] Phillips, W. (2014). *A Composers Guide To Game Music*. MIT Press.

[Procedural Audio Now!, 2015] Procedural Audio Now! (2015). Leonard J PaulWorking and Playing with Procedural Audio - Procedural Audio Now!

[Schweitz, 2010] Schweitz, N. (2010). .mod in Unity – Unity Blog.

[Unity Documentation, 2017] Unity Documentation (2017). Unity - Manual: Tracker Modules.

[Van Nispen Tot Pannerden et al., 2011] Van Nispen Tot Pannerden, T., Huiberts, S., Donders, S., and Koch, S. (2011). The nln-player: A system for nonlinear music in games.

[Velardo, 2017] Velardo, V. (2017). What is Adaptive Music?

[Volk, 2017] Volk, A. (2017). Games: Sound and music for Interactivity and Immersion.